

本书推荐

洪磊与我同事近8年，最早我在原来公司担任CTO，在一次处理系统性能问题时发现洪磊是个疯狂的技术控，那时的他虽然刚工作不久，但是一直在学习一些前沿技术，因此对他印象比较深刻。

后来我自己创业，正巧他也有意向，我们一起一干就是4年。因为是创业公司，我们对框架和平台的选择主要聚焦在开源项目上。工作流平台的选择是个艰难的决定，在对比jBPM5与Tom发起的Activiti后，我决定接受洪磊的建议选择Activiti。

我们公司对这本书的写作极其重视。书中引用的很多案例，都源自实际的项目中的使用经验。在使用过程中也遇到了一些不能满足需求的地方，因此我们自己做了一些改动并通过Github贡献了源代码。目前为止，我们公司是中國地区对Activiti贡献代码次数最多的团队，洪磊是团队的技术核心。这本书也算是对Activiti项目以及对开源代码社区饮水思源的一种表达吧。

今年我们做了一个汽车电商项目——小马购车，初期拿到千万级天使投资，依然选择Activiti作为后台业务系统的流程引擎。事实证明Activiti都能轻松“Hold”住各种应用场景。

希望本书能给大家在Activiti实战中带来更多的帮助。

小马购车CTO

郑国强

咖啡兔同学的《Activiti实战》终于出炉，欣喜之情溢于言表。虽然国内的工作流产品繁多，但是开源一直为Activiti和jBPM垄断。相对来说Activiti延续了一贯方便灵活的特性，又不会在功能上有半分折损，在国内拥有大量的粉丝。咖啡兔同学此前一直致力于Activiti在国内的推广与传播，先后开辟了专栏博客、Activiti论坛网站、QQ群组，并且积极参与Activiti的官方开发，可以说Activiti在国内能达到当前的认知程度，他是功不可没的。可惜，国内尚缺一本可以为Activiti新手答疑解惑，带初阶者更上一层楼的实体书籍。我觉得，这个任务由一直积极活跃于Activiti开源社区，既拥有实际流程项目设计研发经验，又为Activiti官方内核提交过代码的洪磊是再合适不过了。

全书由浅入深地引导读者进入工作流的殿堂，不仅覆盖常见的流程功能与实现方法，还专门提供了作者在实践中总结的经验方法，因此本书必将成为学习流程道路上的得力助手。

小米公司高级软件研发工程师《深入浅出Ext JS》作者 jBPM、Activiti国内推广者

徐会生

在开源BPM领域，你或许不知道“闫洪磊”是谁，但你必须听说过“咖啡兔”，否则，你不能说你曾经玩过开源BPM！

咖啡兔是我从事BPM工作几年来，少有的一位对BPM领域有较深认识的人，同时他对开源界热情的、积极的、无私的奉献精神，让我感动。我们信舟科技SuperBPM平台的诞生，离不开咖啡兔提供的巨大帮助，在此表示感谢。Activiti是一个优秀的项目，让我们能够很容易地将BPM引入我们的企业级应用中，但Activiti毕竟是国外的开源产品，它与国内很多BPM应用还是有些差异的，我们还需要对其做一些必要的个性化扩展和补充，才能用于实际的企业应用。目前国内关于Activiti的专业资料，几乎没有，而《Activiti实战》无疑是目前最佳的入门宝典，书中介绍了很多案例，都是从实际BPM应用中总结的宝贵经验。相信你们将和我一样从中受益，并快速将Activiti集成到自己的企业应用中，让Activiti绽放光芒。

北京信舟科技创始人

袁启勋

自2010年Tom Bayen离开jBoss加入Afresco公司开发出Activit5以来，在短短几年时间内，国内有大批企事业单位和大型金融机构基于Activiti5来构建各类业务流程系统，那是因为Activiti功能稳定、性能良好，对BPMN 2规范完全支持，在API应用友好性和扩展性方面都有着卓越的表现。我从2010年接触工作流至今已15年，认为Activiti5是完全可以与Ultimus、K2等大型商用工作流引擎相媲美的大型开源产品，加上国内有着咖啡兔等众多开源奉献者和非常活跃地社区支持，相信Activiti在国内BPM界会发挥更大地作用和价值。希望这本汇集各种实战经验的书能帮助读者深入了解Activiti。

edoc2CTO Robinson

咖啡兔先生，作为Activiti官方源代码的贡献者，也是Activiti国内传道先锋，对Activiti的背景理念、架构设计、开发运维和使用调优等方方面面的认识、理解和感受，无论从深度还是广度上，都是顶级的。有幸与先生合作，创新地提出分布式平台模式，极大地挖掘和扩展了Activiti的功能和使用范围，更为其谦虚大度的为人、高屋建瓴的思路、团队协作的精神、全面扎实的技术等所震惊，如沐春风，受益匪浅。《Activiti实战》作为先生力作，历时一年有余，全面而富有深度地带领大家认识Activiti的世界，强烈推荐！

银联支付研发中心技术部总经理

王垒

前言

为什么要写这本书

2011年年末，公司承接了一个保险类的业务系统，包含处理核心业务的ERP系统以及日常办公的OA系统，很明显这两种类型的系统都离不开工作流引擎的支持。我用一周时间对比了几个开源的工作流引擎，最后决定使用Activiti作为整套系统的工作流引擎。

现在回想起来，当初的学习过程是多么的“痛苦”啊！当时Activiti才刚满周岁，除了官方提供的尚能看得过去的用户手册之外，再无其他资料可供参考，这对于国内开发者来说尤为痛苦。仅有的用户手册全部都是英文的，为了学习Activiti只能打开翻译软件硬着头皮把手册看了一遍，当然也离不开打入引擎内部的利器——Javadocs。幸运的是，我的第一份工作（3年时间）是为政府单位开发OA系统，这有助于理解在学习Activiti过程中遇到的一些概念性的内容，在此基础上前后花了一周时间写出了第一个在本书中被讲“烂”掉的请假流程。

国内很多技术爱好者都会使用IM软件或论坛建立技术交流社区，也有一些人开设博客撰写相关技术文章。在开始学习Activiti时，很多人都尝试着去寻找这样的社区，结果由于社区规模小、热度不高，常听到学习资料匮乏以及没有成熟的Demo可供参考这样的声音。我喜欢研究技术也乐于分享，从08年就开始以博客的形式分享一些技术学习心得，在基本掌握Activiti的使用方法后就响应社区的号召在GitHub上公开了Activiti入门Demo项目——kft-activiti-demo^[1]，并在个人博客^[2]上发布了几篇与Activiti有关的博文。随着国内使用Activiti的企业越来越多，使得Activiti中文社区活跃度大大增加。最初我要花不少时间回答社区中提出的有关Activiti的问题，长此以往，同一个问题每天要回答多次，而且kft-activiti-demo也太过简单，只能作为入门参考，为了能系统地介绍Activiti，就萌发了撰写一本关于Activiti的书籍的想法。

在2012年6月，机械工业出版社华章公司的首席策划杨福川联系我，表示有意向出版一本Activiti方面的书籍。我们一拍即合，于是就有了这本书，这也让我相信机会是留给有准备的人的。这本书原本预计一年完成，不过由于工作与家庭的原因，在2013年一度中断了大半年，导致这本书的难产。在此也对期待已久的读者说声对不起，同时也是因为你们给予的支持与压力促使我最终完成这本书。

在学习和使用Activiti的过程中也遇到了一些Bug或功能缺陷，例如基本上每个初学者都会遇到的流程图中文乱码问题。对于这些问题最初会通过Bug跟踪系统向官方提交问题，在Activiti的源码从SVN切换到GitHub后就可以很方便地让全球的开发者参与进来。笔者也借助GitHub这个平台为Activiti贡献了一些代码，借此机会也呼吁技术爱好者多多参与开源。

读者对象

本书以“理论 + 实战”的方式引导读者学习，不仅介绍如何使用Activiti，还详细介绍了其遵循的BPMN 2.0规范，所以无论读者是以技术为主还是以业务需求为主，都适合阅读本书。虽然本书中大部分示例都是B/S架构，但不表示Activiti不能在C/S架构中使用。另外，Activiti也不是只针对Java语言的，被其官方定义为BPM平台，借助REST也可以让非Java语言的系统使用Activiti。

适合阅读本书的读者有以下几类：

- Activiti用户和爱好者
- Activiti代码贡献者
- 流程引擎相关的项目经理或者需求人员
- Activiti开发者，或运维人员
- 使用Activiti开发流程平台的公司

如何阅读本书

本书分为四大部分：

第一部分（第1、2章）为准备篇，介绍整个体系结构及其特点，并为后面的内容配置开发环境。

第二部分（第3、4章）为基础篇，介绍两种流程设计器的使用，以及BPMN 2.0规范。

第三部分（第5~14章）为实战篇，本书中内容最多的部分，该部分以实战为主，包括流程定义、流程实例、任务、子流程、多实例、事件以及监听器等。

第四部分（第15~21章）为高级篇，通过集成各种服务、中间件来阐述Activiti不仅是引擎，更是一个BPM平台，最后还深入源码内部剖析Activiti的设计模式及PVM。

勘误和支持

由于笔者的水平有限，加之编写时间仓促，书中难免会出现一些错误或不准确的地方，恳请读者批评指正。为此，特意创建一个在线支持与应急方案的站点<http://www.kafeitu.me/activiti-in-action.html>。大家可以将书中的错误发布在Bug勘误表页面中，同时，在遇到任何问题时，你可以访问其Q&A页面，笔者将尽量在线上为你提供最满意的解答。书中的全部源文件除可以从华章网站^[3]下载外，还可以从笔者提供的这个网址下载，笔者也会将相应的功能更新及时更正出来。如果你有更多的宝贵意见，也欢迎发送邮件至邮箱yanhonglei@gmail.com，期待能够得到你们的真挚反馈。

致谢

首先要感谢Tom Beayens发起了这么优秀的开源项目，以及Alfresco公司Tijs Rademakers带领的Activiti团队对此项目的无偿贡献与支持。

感谢Activiti中文社区中每一位充满激情的朋友——袁启勋、临远、翔子、姜维、王垒、于广岳、第二少、小股东、陈浩、jackyrong、卖字母为生，以及名单之外的更多朋友，感谢他们长期以来对社区的支持和贡献。

感谢机械工业出版社华章公司的编辑杨福川和姜影，我始终坚信专业的事找专业的人做，两位在这两年的时间中从各个方面支持我的写作，他们的鼓励和帮助引导我顺利完成全部书稿。

感谢我的驴友们，和他们一起登山探险让我精神得以放松，希望以后的空余时间我们继续用双脚丈量大地。

最后感谢我的家人，感谢他们对我的支持和鼓励。

谨以此书献给我最亲爱的家人，以及众多热爱Activiti的朋友们！

闫洪磊（咖啡兔）

于上海

[1] <https://github.com/henryyan/kft-activiti-demo>

[2] <http://www.kafeitu.me>

[3] 参见华章网站www.hzbook.com。——编辑注

第一部分 准备篇

工作流（Work Flow）引擎被广泛应用于各种信息化系统中，将原本散乱甚至混乱的业务梳理后制定成业务规范流程，进而约束业务的规范化处理和运转。需求人员、开发人员共同协作制定了符合BPMN 2.0规范的流程定义，之后将其部署到工作流引擎中，由它自动驱动业务流程的进行。

本部分作为准备篇，第1章先介绍了什么是Activiti及其历史背景，然后介绍工作流、BPM、BPMN等概念，使初次接触工作流的读者能快速认识相关概念及各种规范。第2章内容从搭建开发环境开始，之后又介绍了Activiti的Hello World，帮助读者快速入门。

第1章 认识Activiti

很多人对工作流（Workflow）应该不陌生。生活中到处都是活生生的“流”：在单位要请假，首先要找领导审批，在领导审批通过之后申请才获准；从网上购物，下单的那一刻就已经触发了一条工作流，此时可以跟踪购物流程，什么时间下单、什么时间付款、什么时候发货、什么时候收到货，在快递单上签字的时候才等于一条工作流程结束了。

工作流应用广泛，在由任务驱动的各种系统中都能见到它的身影，例如，CRM、ERP、ECM、BI、OA等。在企业应用中还有很多产品或平台集成工作流引擎，用来处理系统运行过程中发起的业务流程。

工作流总是以任务（Task）的形式驱动人处理业务或者驱动业务系统自动完成作业。有了工作流引擎之后，我们不必一直等待其他人的工作进度，直白地说，我们只需要关心系统首页的待办任务数即可，由系统提醒当前有多少待办任务需要处理。

1.1 什么是Activiti

大家第一次接触Activiti的时候不理解它为什么要叫这个名字，从词典中也没有找到对它的解释。可能有人会想到另外一个单词Activity（活动），与Activiti仅一个字母之差。在工作流方面有些基础的读者或许能很快理解，业务流程由多个环节串联起来并且每个环节被赋予任务，而每个任务又可以分为多个活动。举个日常的例子——网上购物的下单环节，首先需要搜索到要购买的商品，然后将其加入到购物车，最后下单填写邮寄地址并付款。这个例子中的每一动作都可以称为活动（Activity），也就是业务流程中最小的组成部分。多个活动在英文中肯定要用复数形式，即Activities；最后以复数化简的方式标示活动的集合，以此来诠释Activiti与工作流的目的与设计。

此项目是Tom Bayen（jBPM创始人）自2010年离开jBoss加入Alfresco公司后的又一力作：第一版在2010年5月发布，当时仅支持最简单的流程处理，之后的版本陆续完善了对BPMN 2.0规范的支持。

值得一提的是，参与项目开发的除了Tom Bayen和十几位核心开发人员之外，还有其他公司的员工参与，例如，SpringSource、MuleSoft、Salves、Signavio、FuseSource、NextLevel等。

Activiti是一个针对企业用户、开发人员、系统管理员的轻量级工作流业务管理平台，其核心是使用Java开发的快速、稳定的BPMN^[1]2.0流程引擎。Activiti是在ApacheV2许可下发布的，可以运行在任何类型的Java程序中，例如服务器、集群、云服务。Activiti可以完美地与Spring集成。同时，基于简约思想的设计使Activiti非常轻量级。

Activiti有着活跃的社区，而且越来越多的企业都选择Activiti作为自己的流程引擎或者将其嵌入到自己的系统平台中（例如ESB）。

接下来简单了解一下工作流及其相关规范的历史。

[1] Business Process Modeling Notation，详细信息可参考<http://www.bpmn.org/>。

1.2 工作流基础

1.2.1 什么是BPM

BPM是Business Process Management的缩写，中文含义是业务流程管理，是一套达成企业各种业务环节整合的全面管理模式。

BPM是为了实现一定的经营目的而执行的一系列逻辑相关的活动的集合。业务流程的输出是满足市场需要的产品或服务。根据功能、管理范围等的不同，企业流程管理一般分为生产流程层、运作层、计划层和战略层四个层次。BPM是根据业务环境的变化，推进人与人之间、人与系统之间，以及系统与系统之间的整合及调整的经营方法与解决方案的IT工具。

BPM最早是由工作流和企业应用集成（Enterprise Application Intergration）逐步融合而发展起来的，当时是为了满足无纸化办公需求（这也是最早的需求之一）。笔者早期参与OA系统开发时曾经见到过“原始的”工作流——没有工作流引擎，整个流程均使用一系列单独为不同任务节点设计的页面串联起来，完成一个节点后在数据库标记当前任务的名称，以此做到“流程驱动”。

随着时间的推移，BPM的定义范围逐步扩展，不仅用来满足无纸化办公需求，现在BPM是一种企业集成技术，作为对面向服务系统架构SOA（Service-Oriented Architecture）、企业应用集成EAI(Enterprise Application Integration)、企业服务总线ESB（Enterprise Service Bus）的补充。

从概念上来说，BPM包含两个不同方面的意思：管理规范 and 软件工程。各大BPM供应商长期以来试图抽象这两个不同的方面，但是依然混乱。

作为管理规范，BPM是每一个战略管理者的责任。BPM是组织必须执行的核心业务流程，包含了企业价值和如何提供其实现。作为日常工作的一部分，业务系统可以借助模型和流程规范地定义业务流程。BPM流程图表达的是执行流程的步骤，已完成特定目标。特别说明的是这些模型用于人与人的沟通。这些都是诠释未决的，这意味着它们可以包含更高级别有价值的信息而不包括不必要的细节。这种诠释未决的过程模型也被称为抽象业务流程（Abstract Business Processes）。

BPM作为软件工程时可以由BPM系统（BPMS）执行可执行的业务流程。可执行的业务流程是在一个流程基础上表示不同的流程顺序。流程图完全可以看做一个抽象的业务流程。可执行流程不同于抽象业务流程，因为它总是以最简单的方式运行。这部分内容也是被大多数厂商认同并接受的。

1.2.2 工作流生命周期

一个完整的工作流生命周期会经过5步，并且迭代循环，如图1-1所示。

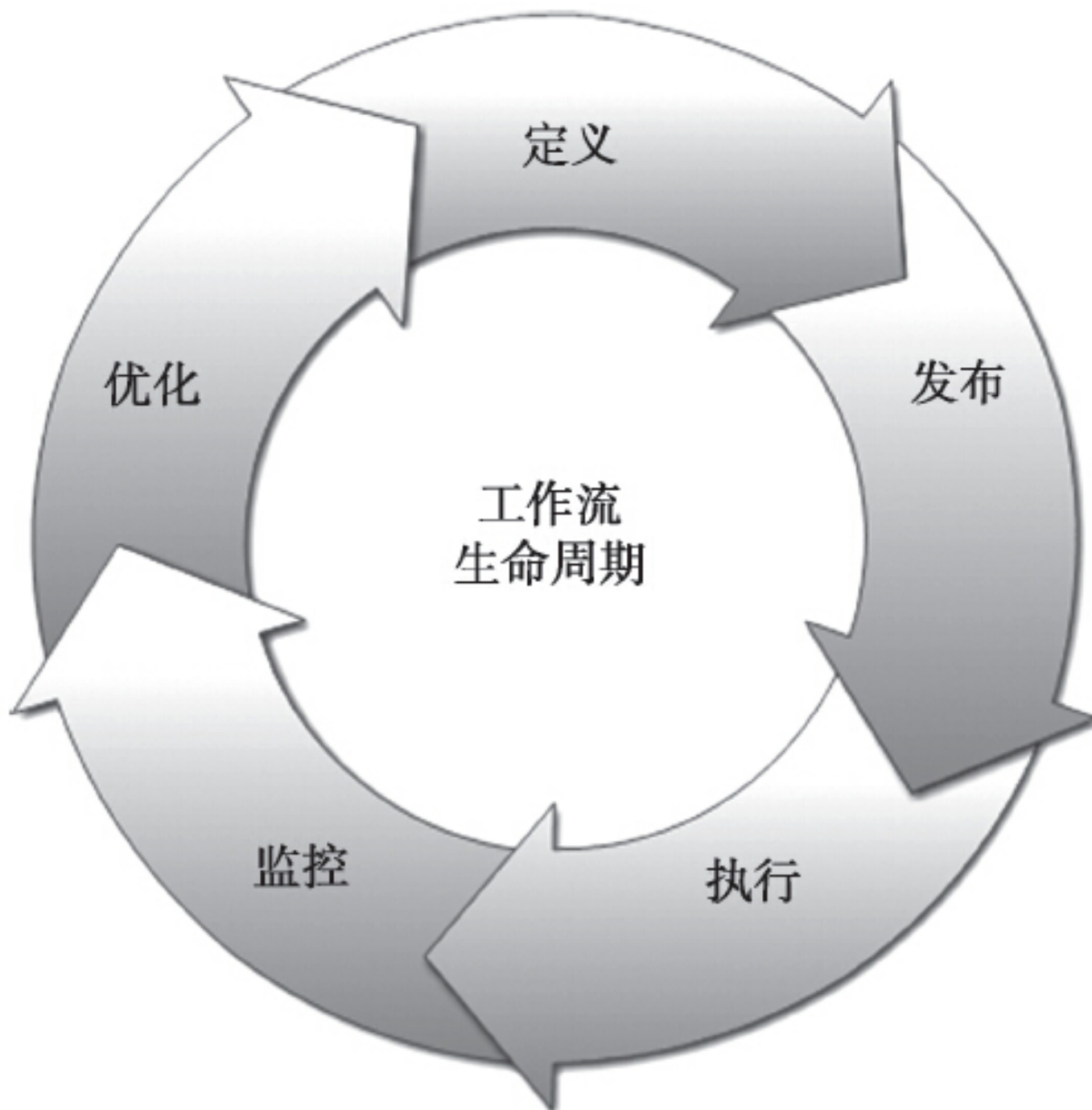


图1-1 完整的工作流生命周期

- **定义**: 工作流生命周期总是从流程定义开始。此阶段的任务主要是收集业务需求并转化为流程定义。一般由业务需求人员进行，然后交由开发人员加工转化为计算机可以识别的流程定义。
- **发布**: 由开发人员打包各种资源，然后在系统管理（平台）中发布流程定义。在具体的流程引擎中包括流程定义文件（bpmn20.xml结尾）、自定义表单、任务监听类。
- **执行**: 具体的流程引擎（例如，Activiti）按照事先定义的流程处理路线以任务驱动的方式执行业务流程。
- **监控**: 此阶段是依赖执行阶段。业务人员在办理任务的同时收集每个任务（Task）的结果，然后根据结果做出相应处理，例如，在采购办公用品流程中，在通过领导审批之后，采购人员就要根据申请单外出采购。
- **优化**: 在此阶段，一个完整的流程已经结束，或许能满足业务需求，或许需要优化，而糟糕的情况是需要重新设计（流程没结束就异常终止），优化与设计正是此阶段需要处理的。根据整个流程的运行过程结果分析问题的根源，然后在此基础上进一步改进，并再次开始一个新的周期。

1.2.3 什么是BPMN

Business Process Modeling Notation，简称BPMN，中文译为业务流程建模标注，是由BPMN标准组织发布的，其第一版BPMN 1.0规范于2004年5月发布。经过多年的改进新的规范BPMN 2.0于2011年发布。之后各大厂商、开源社区均基于2.0规范设计自己的流程引擎，结束了各个厂商“各自为政”的局面，相应地统一了标准，从而有利于以后的产品迁移。

BPMN定义了业务流程图，其基于流程图技术，同时对创建业务流程操作的图形化模型进行了裁减。业务流程的模型就是图形化对象的网图，包括活动（也可以说工作）和定义操作顺序的流控制。

在BPMN 1.x版本中的一些概念，如人工任务、可以执行脚本、自动决策等，都是独立于供应商的可视化标准化的方式。在BPMN 2.0规范中重点聚焦在如何执行语义和一个被业界认可的通用交换格式。这意味着基于BPMN 2.0的流程建模不仅在流程设计器上可以通用，还可以在任何符合BPMN 2.0规范流程引擎上执行。

关于BPMN的细节内容有很多，官方文档足有500多页，本书会在第4章介绍Activiti支持的BPMN 2.0规范以及Activiti在BPMN 2.0规范基础上的扩展。关于BPMN的其他内容本书就不一一列举了，

有兴趣的读者可以仔细阅读官方文档。

1.3 Activiti的特点

1.数据持久化

Activiti的设计思想是简洁、快速。有过应用开发经验的开发人员都知道应用的瓶颈体现在和数据库交换数据的过程中，针对这一点Activiti选择了使用MyBatis，从而可以通过最优的SQL语句执行Command，仅凭如此就能让引擎在速度上保持最高的性能。

2.引擎Service接口

Activiti引擎提供了七大Service接口，均通过ProcessEngine获取，并且支持链式API编程风格。表1-1简单列出七个Service接口及其作用，具体使用会在后面的章节陆续介绍。

表1-1 Activiti引擎的七大Service接口

Service 接口	作 用
RepositoryService	流程仓库 Service，用于管理流程仓库，例如，部署、删除、读取流程资源
IdentifyService	身份 Service，可以管理和查询用户、组之间的关系
RuntimeService	运行时 Service，可以处理所有正在运行状态的流程实例、任务等
TaskService	任务 Service，用于管理、查询任务，例如，签收、办理、指派等
FormService	表单 Service，用于读取和流程、任务相关的表单数据
HistoryService	历史 Service，可以查询所有历史数据，例如，流程实例、任务、活动、变量、附件等
ManagementService	引擎管理 Service，和具体业务无关，主要是可以查询引擎配置、数据库、作业等

3.流程设计器

在BPM4时代有专门的Eclipse插件可以用来设计PDL，同样Activiti团队也专门设计了用来设计BPMN 2.0规范的流程设计器——Eclipse Designer。此外还有Signavio公司为Activiti定制的基于Web的Activiti Modeler流程设计器。

4.原生支持Spring

Activiti原生支持Spring，这一点对企业应用来说尤为重要：可以很轻松地进行Spring集成，非常方便管理事务和解析表达式（Expression）。

5.分离运行时与历史数据

Activiti继承自BPM4，在表结构设计方面也遵循运行时与历史数据的分离，这样的设计可以快速读取运行时数据，仅当需要查询历史数据时再从专门的历史数据表中读取。这种设计方式可以大幅提高数据的存取效率，尤其是当数据日积月累时依然能够快速反应。

1.4 Activiti的应用

目前Activiti在国外已被很多厂商所使用，甚至有人专门成立了公司来培训Activiti的使用。Activiti在国内的发展正在呈直线趋势上升，已经成立了由很多热心的技术爱好者参与的技术社区。目前，很多新项目、新产品都开始采用Activiti作为新一代工作流引擎。

1.在系统集成方面应用

- 与ESB（Enterprise Service Bus，企业服务总线）整合，例如Mule。
- 与规则引擎（Rule Engine）整合，例如JBoss Drools。
- 嵌入已有系统平台，例如，很多公司都开发了自己的系统平台，在其中嵌入Activiti作为平台的一部分。

2.在其他产品中应用

Alfresco^[1]公司的ECM（Enterprise Content Management）产品Alfresco在企业中应用广泛，主要涉及文档管理、协作、记录管理、知识库管理、Web内容管理等。

如果企业或客户正在使用Alfresco管理文档，那么针对文档管理流程设计的流程定义可以直接部署在Alfresco上使用；如果之前没有接触过jBPM而现在学会了使用Activiti，那么不用再去学习其他的流程引擎。关于Activiti与jBPM的区别在1.6节会提到。

在Activiti没有发布之前一直使用jBPM作为流程引擎，在Activiti成熟以后Alfresco同时支持两者，当然在以后的某个时间可能会取消对jBPM的支持。

[1] <http://www.alfresco.com/>

1.5 Activiti架构与组件

Activiti架构中最重要的肯定是引擎，当然还有刚刚提到的外部工具和组件，如图1-2所示。

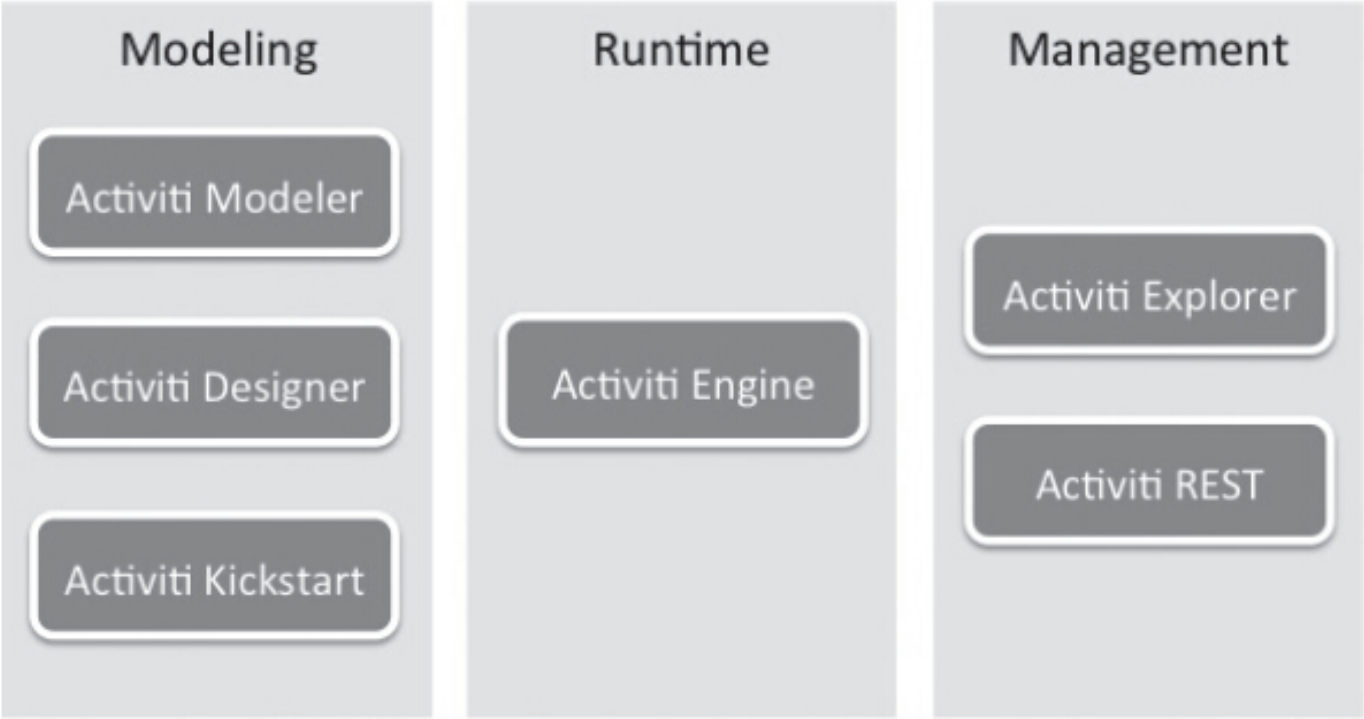


图1-2 Activiti架构图

下面依次介绍Activiti架构图中的各个组件。

- **Activiti Engine**: 作为最核心的模块，提供针对BPMN 2.0规范的解析、执行、创建、管理（任务、流程实例）、查询历史记录并根据结果生成报表。
- **Activiti Modeler**: 是模型设计器，其并非由Activiti公司所开发，而是由业界认可的Signavio公司赠送的（Signavio^[1]原本是收费的产品，现在被免费授权给Activiti用户使用）。适用于业务人员把需求转换为规范流程定义。
- **Activiti Designer**: 功能和Activiti Modeler类似，同样提供了基于BPMN 2.0规范的可视化设计功能，但是目前还没有完全支持BPMN规范的定义。适用于开发人员，可以把业务需求人员用Signavio设计的流程定义（XML格式）导入到Designer中，从而让开发人员将其进一步加工成为可以运行的流程定义。
- **Activiti Explorer**: 可以用来管理仓库、用户、组，启动流程、任务办理等。此组件使用REST风格API（目的在于让开发人员快速入门），提供一个基础的设计模型。如果业务简单，也可以直接使用无需开发。还可以作为后台管理员的流程、任务管理系统使用。
- **Activiti REST**: 提供Restful风格的服务，允许客户端以JSON的方式与引擎的REST API交互，通用的协议具有跨平台、跨语言的特性。

[1] <http://www.signavio.com/en.html>

1.6 Activiti与jBPM5比较

目前流行的工作流引擎有Activiti和jBPM5，而在jBPM5发布以前大多数项目、平台都是基于jBPM3、jBPM4开发的。本节内容从技术和实际应用上对Activiti和jBPM5进行比较。表1-2从技术层面比较了两者的区别。

表1-2 Activiti与jBPM5的技术层面对比

技术组成	Activiti	jBPM
ORM 框架	MyBatis3	Hibernate3
持久化标准	无	EJB JPA 规范
事务管理	MyBatis 自带 /Spring 集成事务	Bitronix，基于 JTA 事务管理
数据库连接方式	Jdbc/DataSource	Jdbc/DataSource
Spring 支持	原生支持 Spring，在流程中可以使用 Spring 代理的 Bean 作为表达式的一部分，并且支持 JPA 及事务管理	默认没有提供对 Spring 的支持
支持的数据库	Oracle、SQL Server、MySQL、H2、内存数据库等	Oracle、SQL Server、MySQL、内存数据库等
设计模式	命令模式、观察者模式等	
内部服务通信	Service 间通过 API 调用	基于 Apache Mina 异步通信
集成接口	SOAP、Mule、RESTful	消息通信

(续)

技术组成	Activiti	jBPM
支持的流程格式	BPMN2、xPDL、jPDL 等（由 PVM 实现）	目前仅只支持 BPMN2 xml
引擎核心	PVM（流程虚拟机）	Drools
技术前身	jBPM3、jBPM4	Drools Flow
团队成员	除了 Alfresco 公司的雇员之外还有 Spring-Source、MuleSoft、Salves、Signavio、FuseSource、NextLevel 等公司的员工加入	有一个专门的团队，此外还有一些个人参与者
附加工具	提供了基于 Eclipse 插件的流程设计器——Eclipse Designer，提供基于 REST 风格的 Activiti Explorer，可以用来管理仓库、用户、组、启动流程、任务办理等	同样提供 Eclipse 插件和一个 Web 应用管理流程
发布周期	固定每两个月发布一版，其中包括：引擎、Eclipse Designer、Activiti Explorer、REST 应用	jBPM 的发布周期相对来说不太固定，发布内容包括引擎及基于 Eclipse 的设计器

Activiti是基于jBPM4设计的衍生版本，如果选择Activiti可以继续沿用jBPM的思想理念设计、整合Activiti到项目或平台中，这也是相对于jBPM5来说的一个优势；相反，对于jBPM5来说要花点时间重新接受开发者的设计思想。

在各个流程引擎社区中有很多关于该如何选择Activiti和jBPM5的讨论，这两者有着很多相似的地方，争论主要是对规则引擎的支持：jBPM5是基于Drool Flow所有自然深度继承而来的规则引擎Drools；早期的Activiti功能比较简单，后来陆续添加的新特性也支持规则引擎Drools，开发人员只要简单配置规则接口即可达到与jBPM5一样的效果。

1.7 本章小结

本章内容主要是以初识工作流和Activiti的角度去讲解什么是工作流、什么是Activiti。从Activiti项目的发起、特点、应用、架构，以及与其他同类产品比较的角度在概念层给读者一个引导和认识。作为一个开发人员，能够知道为什么需要学习Activiti，Activiti能帮助企业解决什么问题，为什么要选择Activiti而不是其他的工作流引擎。看到这里你或许会不耐烦了，会想起一句话：“Talk is cheap.Show me the code^[1]”。下一章将带领你体验Activiti的Hello World。

[1] 这是Linux的作者Linus Torvalds的名言，意思是：能说算不上什么，有本事就把你的代码给我看看。

第2章 搭建Activiti开发环境

第1章介绍了关于工作流的概念及Activiti的发起背景，带领大家大体了解Activiti能帮助企业解决什么问题，作为一个开发人员为什么需要学习Activiti，还简单介绍了一下Activiti的架构以及与其他工作流引擎的区别。本章将带领你一步一步搭建开发环境，并运行期待已久的Hello World程序。

2.1 下载Activiti

通过浏览器访问页面：<http://activiti.org/download.html>，其中列出自Activiti发布以来历次版本的压缩包和相关文档（10分钟入门、用户手册以及JavaDoc）。

在“Latest Release”处下载下面的压缩包，在笔者写作本书时最新版本为Activiti 5.9。细心的读者可能注意到：在“Older releases”的发布列表中5.6版本之前基本上是一个月发布一版，从5.6版本之后基本固定为2~3个月发布一版。

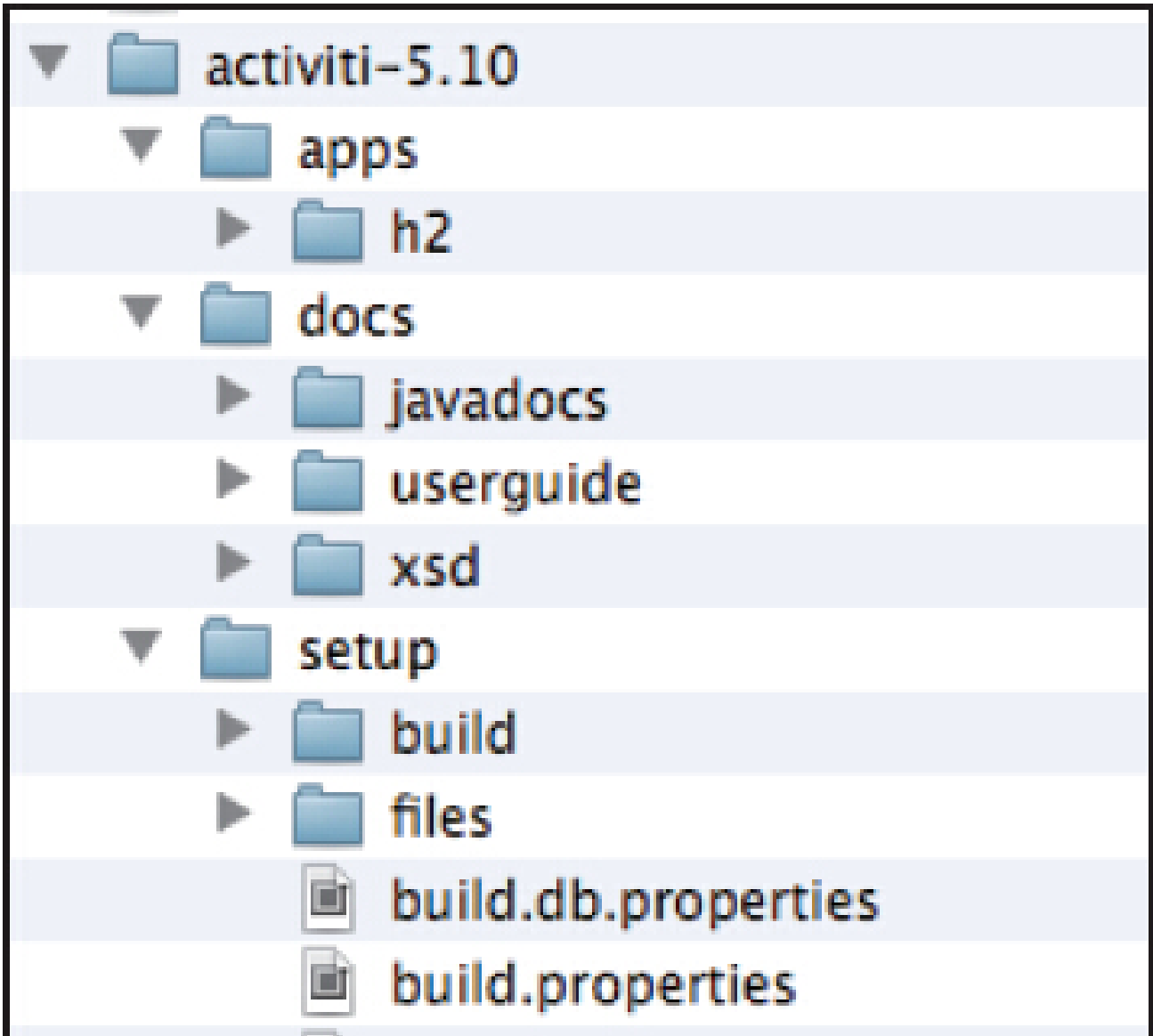
 **说明** 初学者很难理解的地方是：为什么Activiti的版本从5.0开始？在第1章的内容中提到过Activiti是基于jBPM4开发的，所以就直接版本号累加，最终从5.0版本开始。

2.1.1 目录结构

在最初撰写本章内容时最新的版本为5.9，目录结构和下一版本（5.10）一致；但是从5.11版本开始发生了比较大的变化，所以分两个小部分说明两者在目录结构上的不同。本书完稿时最新版本为5.16.3。

1.5.10及之前的版本

- 对于Activiti 5.10及之前的版本，解压压缩包之后目录结构如图2-1所示。
- 下面我们来详细介绍图2-1中的目录结构。
- docs，该目录包含了三种文档：javadocs、userguide、xsd。
 - javadocs：包名按照功能模块划分，org.activiti.engine.*，接下来在2.1.2节中详细介绍。
 - userguide：用户手册，包含环境配置、10分钟快速入门，以及各个功能模块的使用教程。
 - xsd：包含BPMN 2.0规范的XSD文件，以及Activiti扩展的自定义标签XSD。
 - setup：用来构建、启动Activiti Explorer演示程序。通过ant demo.start命令即可自动下载tomcat，配置数据库，最后打开浏览器访问应用。具体操作将在2.4节详细介绍。
 - workspace：该目录包含了各种应用的实例程序，都以单元测试的形式展示功能的使用方式。读者可以直接将项目导入Eclipse查看其源代码，从而通过断点调试来学习。



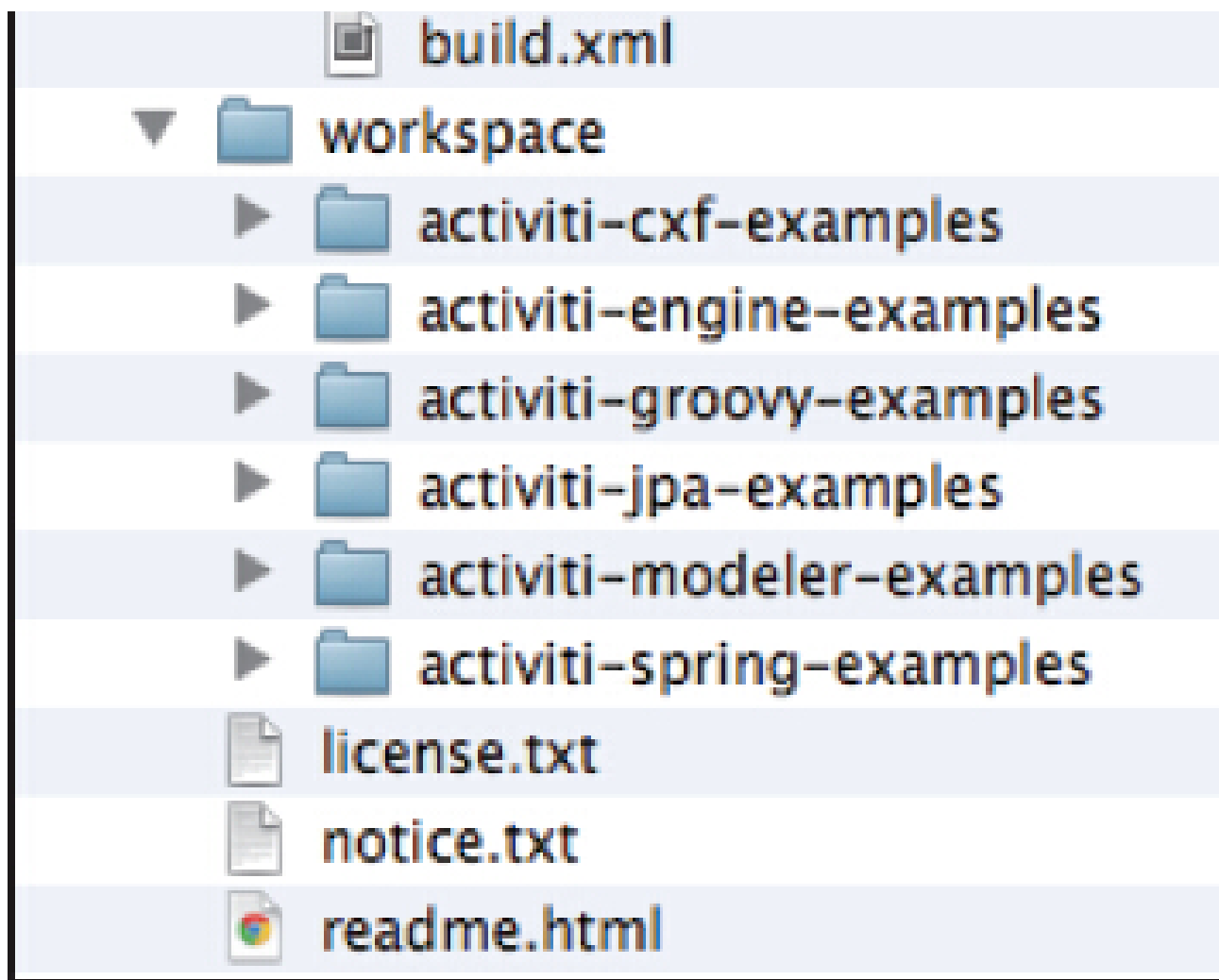


图2-1 Activiti 5.10及之前版本的目录结构

2.5.11及之后的版本

对于Activiti 5.11及之后的版本，解压压缩包之后目录结构如图2-2所示。

下面我们来详细介绍图2-2中的目录结构。

- Database，该目录包含了针对Activiti引擎表的创建（create）、删除（drop）及版本升级（upgrade）三种类型的脚本：创建、删除类型的脚本以“activiti.oracle.[create|drop].[identity|engine|history].sql”这样的结构命名SQL脚本文件；升级脚本以“activiti.oracle.upgradestep.[5x].to.[5(x+1)].history”的结构命名SQL脚本文件，其中“x”表示小版本号。
- docs，该目录也包含了三种文档：javadocs、userguide、xsd。
 - javadocs：包名按照功能模块划分，org.activiti.engine.*，接下来在2.1.2节中详细介绍。
 - userguide：用户手册，包含环境配置、10分钟快速入门，以及各个功能模块的使用教程。
 - xsd：包含与流程定义相关的scheme，例如，BPMN 2.0、BPMNDI及Activiti在BPMN 2.0规范上扩展的元素。
- libs：相比5.10之前的版本移除了第三方的依赖，仅仅包含了Activiti引擎的各个模块（engine、bpmn-converter、bpmn-model、camel、cdi、cxf、json-converter、modeler、mule、osgi、rest、spring）的class文件以及源码。这么做的目的是通过Maven管理依赖（在2.3.2节介绍了如何使用Maven管理依赖）避免因Jar版本不同导致冲突（大部分开源项目及国外的项目都使用Maven管理依赖，所以强烈建议没接触过Maven的读者花点时间学习使用它）。
- wars：从5.11版本开始对explorer模块和rest模块进行了拆分，使得rest模块可以独立运行；运行explorer的方式也随之发生了变化，不再使用ant脚本（2.5.1节）运行，而是提供一个独立的war包，自行部署tomcat或jboss等Web服务器中。

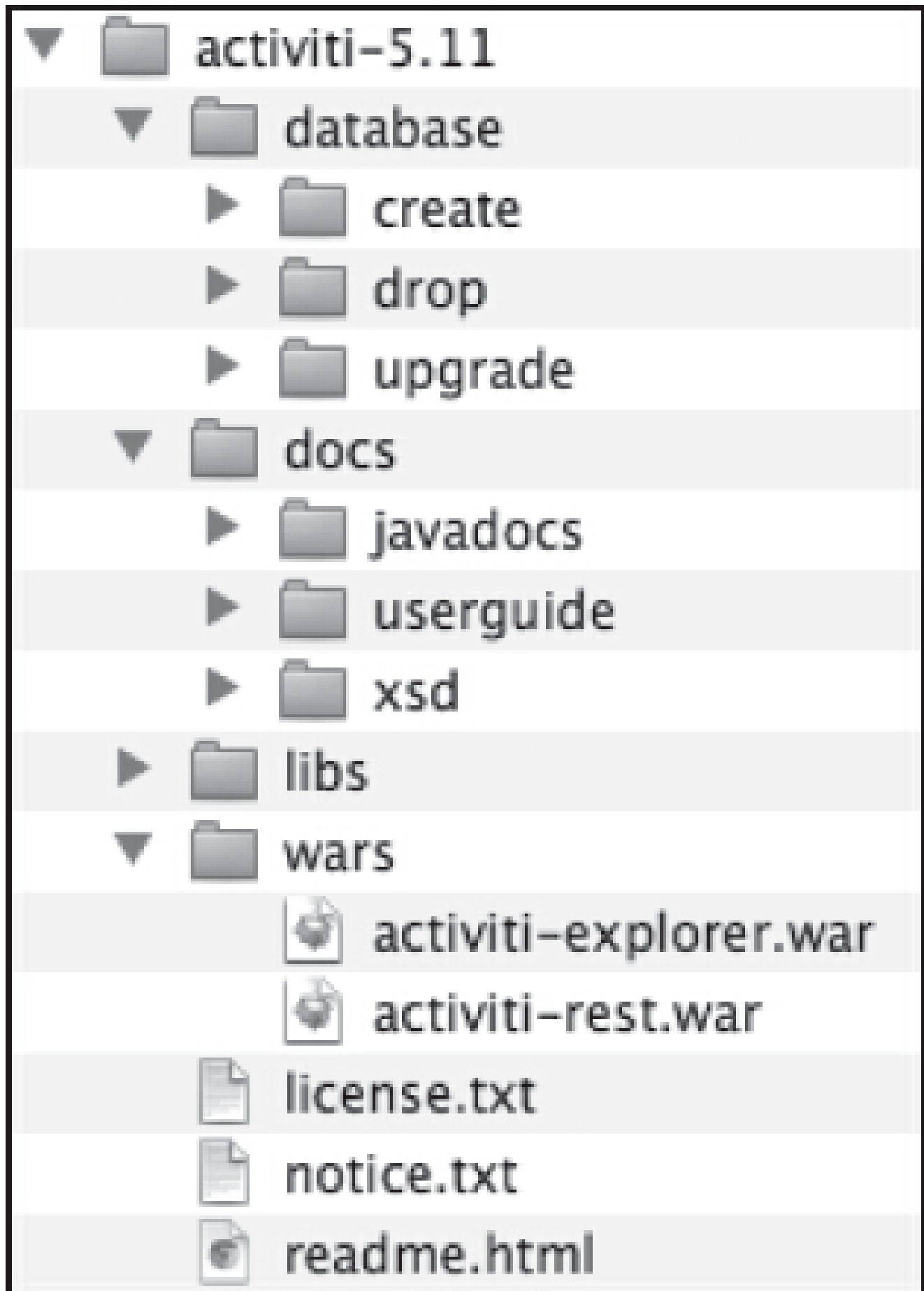


图2-2 Activiti 5.11及之后版本的目录结构

2.1.2 Javadocs

Javadocs是开发人员与Activiti交流的“语言”，在开始编写第一个Hello World之前先来大致了解一下每个包（也就是module）的功能及各个接口的作用。当然不太想了解这些内容的读者也可以暂时

跳到这里，直接开始搭建环境并编写第一个Hello World程序，最后再回过头来了解。

Javadocs一共包含11个package，下面依次介绍它们的作用范围。

- 1) `org.activiti.engine`: 包含上一章提到的七大类Service接口、异常类定义和流程引擎（Process Engine）及流程引擎配置（Process Engine Configuration），另外还定义了一些运行时（Runtime）异常类。
- 2) `org.activiti.engine.delegate`: 该包定义了处理流程的行为、监听事件的规范。在实际应用中可以在流程定义中配置实现了监听接口的类处理业务逻辑，例如可以在流程结束时由系统自动归档。在第1章中提到过Activiti使用的是监听者（Observer）模式，在流程运行过程中（任务完成）引擎会遍历注册的监听并依次调用。
- 3) `org.activiti.engine.form`: 该包应用在内置表单的场景下，在一些企业或客户要求自定义表单的需求中使用。定义表单有两种方式：第一种是直接流程定义中设置每个节点的表单内容，可以设置每个字段（Field）的类型、是否可以编辑等属性；另外一种就是通过外置表单的形式，通过formkey指定外置表单文件的名称，类型可以是.xml或.form。表单的读取、提交均可以通过FormService接口完成。
- 4) `org.activiti.engine.history`: 该包包含了历史记录查询对象及查询结果的历史数据对象接口。可以查询历史流程实例（HistoricProcessInstance）、历史任务（HistoricTask）、历史活动（HistoricActivity）、历史详细（HistoricDetail）等。在13.3节中，流程跟踪功能就是通过HistoryService接口进行查询来实现的。
- 5) `org.activiti.engine.identity`: 该包可以用来管理身份和认证，功能依托IdentityService接口。在这里需要说明一下，Activiti有自己的Identity模块，因为流程中的用户任务（userTask）需要指派给某个人或某个角色。关于任务指派在后面12.1.3节会介绍。讲到这里顺便提一下，关于身份认证管理功能和企业中已有身份认证管理模块的冲突解决办法将在第19章中详细讨论。
- 6) `org.activiti.engine.management`: 该包主要是用来实现针对流程引擎的管理功能，通过调用接口ManagementService可以监控引擎状态、任务调度、数据库数据读取。
- 7) `org.activiti.engine.query`: 该包没有具体的功能，定义了查询的共有特性。细心的读者在查看API的时候会发现XxxQuery接口均继承自Query，并且使用了JDK 5提供的泛型。
- 8) `org.activiti.engine.repository`: 该包包含了针对流程资源的管理与查询。依托RepositoryService接口可以部署流程定义、自定义表单、规则等文件，还可以读取流程图片、流程定义（bpmn20.xml）文件。
- 9) `org.activiti.engine.runtime`: 在第1章中提到了Activiti把流程数据划分为两种：运行时数据和历史数据。刚刚介绍了org.activiti.engine.history包依托HistoryService查询历史数据，相应地，也可以通过RuntimeService接口查询运行时数据，例如可以查询出当前用户的待签收任务、待处理任务，以及正在处理中的流程实例对象。当然除了查询数据之外还要启动流程，然后得到实现了ProcessInstance接口的实例，从而在业务中根据流程实例继续处理业务信息。另外在Activiti 5.9版本中添加了对流程定义状态的控制功能：挂起和恢复。
- 10) `org.activiti.engine.task`: 该包包含任务对象的定义，依托TaskService接口可以对任务（Task）全面管理，例如，任务创建、删除、任务指派、批注管理、附件管理以及变量查询。在12章将会逐一介绍TaskService操控Task。
- 11) `org.activiti.engine.test`: 顾名思义，该包针对快速创建测试用例提供基类和注解（Annotation）。现在已经有很多公司或开发人员使用测试驱动（Test Driven Development，TDD）方式编码，Activiti提供的这一功能大大方便了验证流程定义中业务逻辑正确与否的效率。此包不仅提供了测试基类，还支持注解（Annotation）方式自动部署流程定义。本书的很多示例也将先以TDD方式引导读者学习。

2.2 环境配置检查

在准备搭建开发环境之前需要读者检查自己的系统环境是否已安装JDK、Ant和Maven，如果已经安装，那么需要检查版本是否满足Activiti的最低要求。

2.2.1 检查并安装JDK

在Activiti 5.10版本之前要求JDK的最低版本为JDK 1.5（即5.0），从Activiti 5.11版本开始要求最低JDK为1.6（即6.0）；如果本地配置低于不同版本的最低要求，那么需要到Oracle官方下载（<http://www.oracle.com/technetwork/java/javase/downloads/index.html>）并安装。

在检查JDK前先确认JAVA_HOME已正确设置。

1) Windows用户：

```
C:\Documents and Settings\henryyan> echo %JAVA_HOME%
```

```
C:\Documents and Settings\henryyan> java -version
```

2) Linux、Mac OS用户：

```
henryyan@hy-hp ~$ echo $JAVA_HOME
```

```
henryyan@hy-hp ~$ java -version
```

笔者的Java环境如图2-3所示。

```
→ henryyan@hy-hp ~$ java -version
java version "1.6.0_30"
Java(TM) SE Runtime Environment (build 1.6.0_30-b12)
Java HotSpot(TM) 64-Bit Server VM (build 20.5-b03, mixed mode)
```

图2-3 在Linux系统中安装配置Java环境

2.2.2 检查并安装Ant

Activiti Explorer在1.5节已经提到了，可以快速运行示例程序。示例程序需要使用Ant构建运行。

官方要求Ant的版本为1.8.1或更高，如果本地没有Ant安装软件，那么先从Apache网站下载 (<http://ant.apache.org/bindownload.cgi>)，然后将其解压并安装到本地目录。

检查Ant前先确认ANT_HOME是否已正确设置。

1) Windows用户：

```
C:\Documents and Settings\henryyan>echo%ANT_HOME%
```

```
C:\Documents and Settings\henryyan>ant-version
```

2) Linux、MacOS用户：

```
henryyan@hy-hp~>echo$ANT_HOME
```

```
henryyan@hy-hp~>ant-version
```

笔者的Ant环境如图2-4所示。

```
→ henryyan@hy-hp ~ echo $ANT_HOME
/opt/devtools/ant/apache-ant
→ henryyan@hy-hp ~ ant -version
Apache Ant(TM) version 1.8.4 compiled on May 22 2012
```

图2-4 在Linux系统中安装配置Ant环境

2.2.3 检查并安装Maven

Maven作为一款优秀的构建工具（不仅仅是构建）已经被全球很多的开发者、开源组织使用，近年来也被国内越来越多的公司和组织使用。未接触Maven或不太熟悉相关内容的读者可以参考国内Maven专家许晓斌著的《Maven实战》。^[1]

Activiti引擎的源码也是使用Maven构建的，并且公开Maven仓库方便开发者下载引擎。本书所有示例程序都将使用Maven管理依赖，所以读者一定要检查Maven的配置是否正确。

如果还未安装Maven，那么先从Apache网站下载 (<http://maven.apache.org/download.html>)，然后将其解压到本地目录。

检查Maven要先确认MVN_HOME是否设置。

1) Windows用户：

```
C:\Documents and Settings\henryyan>echo%MVN_HOME%
```

```
C:\Documents and Settings\henryyan>mvn-v
```

2) Linux、Mac OS用户：

```
henryyan@hy-hp~>echo$MVN_HOME
```

```
henryyan@hy-hp~>mvn-v
```

笔者的Maven环境如图2-5所示。

```
→ henryyan@hy-hp ~ echo $MVN_HOME
/opt/devtools/maven/apache-maven
→ henryyan@hy-hp ~ mvn -v
Apache Maven 3.0.4 (r1232337; 2012-01-17 16:44:56+0800)
Maven home: /opt/devtools/maven/apache-maven
Java version: 1.6.0_30, vendor: Sun Microsystems Inc.
Java home: /opt/devtools/jdk/jdk1.6.0_30/jre
Default locale: en_US, platform encoding: UTF-8
OS name: "linux", version: "3.2.0-24-generic", arch: "amd64", family: "unix"
```

图2-5 在Linux系统中安装配置Maven环境

[1] <http://www.juvenxu.com/mvn-in-action>

2.3 配置文件介绍

本节来介绍两个配置文件，一个是Maven的pom.xml文件，另外一个就是Activiti的默认配置文件activiti.cfg.xml。

2.3.1 Activiti配置文件

在bpmn20-example工程的src/test/resources中有一个activiti.cfg.xml文件，此文件就是Activiti的配置文件，用来定义引擎初始化参数、bean、邮件服务器及各种监听器。

代码清单2-1展示了一个标准的Activiti配置文件。

代码清单2-1 标准的Activiti配置文件

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd">
    #1
    <bean id="processEngineConfiguration"
          class="org.activiti.engine.impl.cfg.StandaloneInMemProcessEngineConfiguration">
        #2
        <property name="databaseType" value="h2" />
        <property name="databaseSchemaUpdate" value="true" />
        <property name="jobExecutorActivate" value="false" />
        <property name="history" value="full" />
    </bean>
</beans>
```

在第1章中提到过Activiti和Spring无缝集成，熟悉Spring的读者一眼就看出这就是Spring的配置文件。是的，在XML的命名空间中定义了spring-bean，即<http://www.springframework.org/schema/beans>。

- #1处定义了一个id为processEngineConfiguration的bean对象，其中processEngine-Configuration即为Activiti默认的引擎配置管理器名称。
- #2处指定了一个具体的Java类，由Spring负责实例化引擎配置管理器并注入#3处的一系列配置参数。此处的引擎配置管理器是基于内存数据库的，因为其速度快，常用于测试环境。
- #3处的配置参数含义如表2-1所示。

表2-1 Activiti引擎配置管理器的参数及其含义

属 性 名 称	属 性 说 明
databaseType	数据库类型，默认为 H2 。当使用非 H2 数据库时需要特别声明，例如在生产环境中一般不会使用 H2 作为数据库。目前 Activiti 支持的数据库有：H2、MySQL、Oracle、Postgres、MSSQL、DB2
databaseSchemaUpdate	用来声明数据库脚本更新策略，和 hibernate 的机制类似。取值如下： ☐ false ，什么都不做 ☐ true ，当 Activiti 的表不存在时自动创建；当 Activiti 的 jar 文件中定义的版本号与数据库中记录版本号不一致时会自动执行相应的升级脚本，并且会记录升级过程 ☐ create-drop ，创建引擎时执行初始化脚本，引擎销毁时执行删除数据库脚本
jobExecutorActivate	用来设置是否启用作业执行功能，默认为 false 。在将该值设置为 true 之后，引擎会不间断地刷新数据库的作业表，检查是否存在需要执行的作业，有则触发作业的执行。作业的来源有多种，例如各种时间事件或异步任务执行
history	用来设置记录历史的级别，默认为 audit 。支持以下几种类型： ☐ none ，不保存任何历史记录，可以提高系统性能 ☐ activity ，保存所有的流程实例、任务、活动信息 ☐ audit ，也是 Activiti 的默认级别，保存所有的流程实例、任务、活动、表单属性 ☐ full ：最完整的历史记录，除了包含 audit 级别的信息之外还保存详细，例如流程变量、表单属性

2.3.2 Maven配置文件

讲解Maven配置文件主要是针对没有接触过Maven或对Maven不太熟悉的读者。

打开bpmn20-example中的pom.xml查看文件内容。下面以代码片段的形式说明Maven配置文件的作用。

要在Maven项目中使用一个依赖只需要声明groupId、artifactId、version三个属性即可，具体的XML如下：

```
<dependency>
  <groupId>org.activiti</groupId>
  <artifactId>activiti-engine</artifactId>
  <version>${activiti.version}</version>
</dependency>
```

- 其中：
- **groupId**，一般以公司或组织的域名倒序定义；
 - **artifactId**，一个组件的ID标识；
 - **version**，依赖的版本，此处用一个占位符代替实际的版本号。

version标签中的\${activiti.version}表达式可以在properties标签中定义属性的值，如下：

```
<properties>
```

```
<activiti.version>5.10</activiti.version>
</properties>
```

通过指定activiti.version为5.10，Maven就知道需要从仓库下载版本为5.10的Activiti对应的jar文件，还可以下载依赖组件的javadocs或sources等。

Maven在需要下载依赖的时候默认从中央仓库搜索，当匹配到存在的组件时将其从远程下载到本地仓库。但是，当中央仓库不存在某个组件的时候就需要定义第三方仓库供Maven查询、下载。目前Activiti还未被收录到中央仓库，所以我们需要指定一个第三方仓库，即Activiti提供的仓库，告诉Maven当找不到依赖时从此仓库查询。以下代码定义了Activiti的公开仓库（从5.14版本开始，可以从Maven中央仓库直接下载）：

```
<!-- Maven仓库定义 -->
<repositories>
  <!-- Activiti的仓库 -->
  <repository>
    <id>Activiti</id>
    <url>http://maven.alfresco.com/nexus/content/repositories/activiti</url>
  </repository>
</repositories>
```

以此类推，当日常开发中遇到某些依赖组件不存在的情况时，可以先查看官方网站是否提供了Maven仓库，如果有则加入到repositories中，否则只能用私服的方式（私服如何使用不在本书讨论范围内，读者可以参考《Maven实战》一书）。

2.4 Hello World

Hello World如此经典，几乎每学习一门新技术都从它开始，它是一把打开技术之门的万能钥匙，因为通过Hello World^[1]能快速了解一门技术如何配置、运行，以及得到什么样的结果。

好吧，让我们一起开启探索Activiti的大门。

2.4.1 最简单的流程定义

代码清单2-2是一个最简单的请假流程定义文件，简单到仅有开始节点和结束节点。

代码清单2-2 最简单的请假流程定义文件leave.bpmn

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL" #1-S
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:activiti="http://activiti.org/bpmn"
  xmlns:bpmndi="http://www.omg.org/spec/BPMN/20100524/DI" xmlns:cmgdc="http://www.omg.org/spec/DD/20100524/DC"
  xmlns:cmgdi="http://www.omg.org/spec/DD/20100524/DI" typeLanguage="http://www.w3.org/2001/XMLSchema"
  expressionLanguage="http://www.w3.org/1999/XPath" targetNamespace="http://www.kafeitu.me/activiti-in-action" #1-E
  <process id="leave" name="Leave" #2
    <startEvent id="startevent1" name="Start"></startEvent> #3
    <endEvent id="endevent1" name="End"></endEvent> #4
    <sequenceFlow id="flow1" name="" sourceRef="startevent1" #5
      targetRef="endevent1"></sequenceFlow>
  </process>
```

在解释这个文件的内容之前我们先来了解一下为了更好地理解代码含义本书所做的一些约定。

 **说明** #号后面加数字用来表示代码清单的第几处（不是行号），其中1-S和1-E分别代表第1处标记的开始（Start）和结束（End）。

在详细讲解流程定义文件之前还要来看看对应的图片形式的流程定义，如图2-6所示。

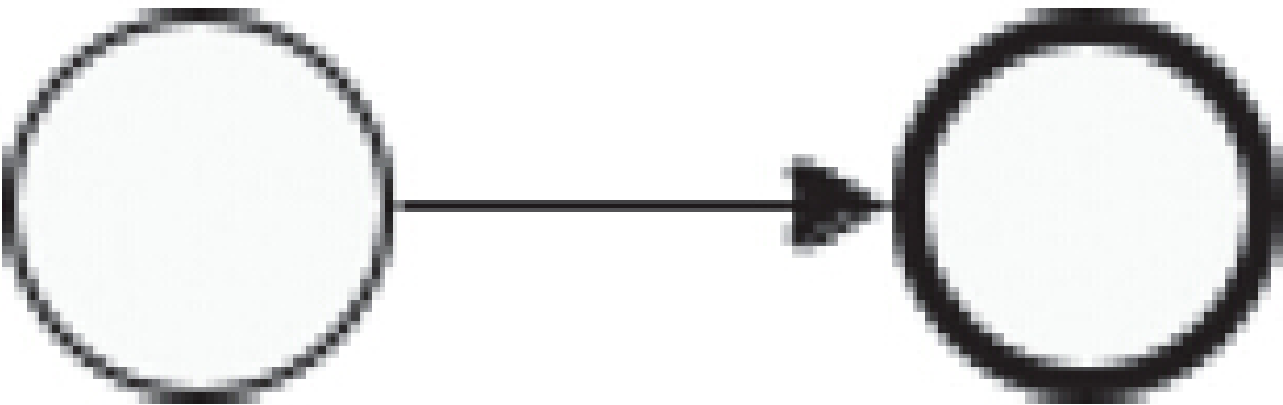


图2-6 最简单的流程定义

在代码清单2-2中，1-S处的definitions标签表示BPMN 2.0规范中定义的开始，可以包含多个process标签；紧接着是XMLSchema的定义，用来验证XML内容是否符合规范；最后一个属性targetNamespace是必需的，用来声明命名空间。为什么必须定义targetNamespace呢？命名空间是BPMN 2.0规范为了易于区分、归类流程定义所设，其值可以是任意文字，当然一般由公司或组织的名称定义，也可以更具体到一个项目，在本例中使用的是<http://www.kafeitu.me/activiti-in-action>，本书所有的targetNamespacet也均使用来声明命名空间。

代码清单2-2中的#2处，定义了id属性为leave的process标签，以此来标记这是一个请假流程定义的开始。

代码清单2-2中的#3处，定义了流程的入口，即空启动事件startEvent，当流程启动时总是以startEvent开始。

代码清单2-2中的#4处，定义了流程的唯一结束出口，即空结束事件endEvent，流程线（flow）执行到endEvent时表示流程结束。可以定义多个结束事件。

代码清单2-2中的#5处，定义了一个sequenceFlow标签，用来描述各个流程节点之间的关系。图2-6中的箭头线就是sequenceFlow所定义的部分。sequenceFlow用sourceRef表示从哪里开

始“流”到哪里。

2.4.2 创建单元测试类

先来看看我们的Java类代码，如代码清单2-3所示，让程序运行起来，然后再解释代码的含义。

代码清单2-3 最简单的请假流程的Java类

```
public class VerySimpleLeaveProcessTest {
    @Test
    public void testStartProcess() throws Exception {
        // 创建流程引擎，使用内存数据库
        ProcessEngine processEngine = ProcessEngineConfiguration #1-S
        .createStandaloneInMemProcessEngineConfiguration()
        .buildProcessEngine(); #1-E
        // 部署流程定义文件
        RepositoryService repositoryService = processEngine.getRepositoryService(); #2
        repositoryService.createDeployment() #3-S
        .addClasspathResource(
            "me/kafaitu/activiti/helloworld/sayhelloleave.bpmn.xml").deploy(); #3-E
        // 验证已部署流程定义
        ProcessDefinition processDefinition = repositoryService #4-S
        .createProcessDefinitionQuery().singleResult();
        assertEquals("leavesayhello", processDefinition.getKey()); #4-E
        // 启动流程并返回流程实例
        RuntimeService runtimeService = processEngine.getRuntimeService(); #5
        ProcessInstance processInstance = runtimeService #6-S
        .startProcessInstanceByKey("leavesayhello");
        assertNotNull(processInstance);
        System.out.println("pid=" + processInstance.getId() + ", pdid="
            + processInstance.getProcessDefinitionId()); #6-E
    }
}
```

下面对代码清单2-3中所有标记处的作用依次进行解释。

#1-S至#1-E是通过编程方式创建一个流程引擎实例，即通过ProcessEngineConfiguration工具类的createStandaloneInMemProcessEngineConfiguration()方法创建一个使用H2内存数据库的流程引擎实例，默认的jdbcUrl为jdbc:h2:mem:activiti。当然，除了此方法之外还有其他创建引擎实例的方法，例如调用ProcessEngineConfiguration.createXXX().buildProcessEngine()，在调用过程中还可以通过编程方式配置引擎的参数ProcessEngineConfiguration.createXXX()http://www.hzcourse.com/resource/readBook?path=/openresources/teach_ebook/uncompressed/15030/OEBPS/Text/..setFoo(argument).buildProcessEngine()。

#2处紧接着使用刚刚创建的引擎实例获取RepositoryService，第1章列出的Activiti的七大Service接口都可以由ProcessEngine通过getXxxService()方法获取。

#3处使用RepositoryService部署位于classpath中的流程定义文件sayhelloleave.bpmn。

#4-S至#4-E处用来验证刚刚部署的流程是否成功。这里需要说明一下，在通过七大Service接口查询对象时均使用xxxService.createXxxQuery()方式创建查询对象。

#5处和#2处一样，通过引擎实例获取RuntimeService对象。

#6-S处使用runtimeService启动一个流程并返回流程实例。此外runtimeService和创建流程引擎的方式类似，也提供了多种启动流程的方式，可以使用runtimeService.startProcessInstanceXxx()启动流程实例。在启动的同时还可以设置流程变量，具体使用方法可以先参考API文档中的方法说明，以后的章节会陆续讲到根据不同的需求使用不同的启动方式。接下来验证流程是否启动成功，这仅是简单的null验证。读者可以自行扩展验证，以验证自己的猜测结果，这也是一种很好的学习方式。最后输出已启动的流程实例的ID和流程定义的ID。

2.4.3 运行Hello World

可以将项目bpmn20-example导入至IDE，笔者使用的是Eclipse IDE for Java EE Developers(Indigo)，选择此版本是考虑到后续章节中有基于Web的应用。本书的例子均使用Maven管理，在实例中已配置好依赖需要使用的仓库（Repository），通过<http://www.eclipse.org/m2e/download/>安装M2Eclipse插件，然后单击菜单：File->Import，选择对话框中的Maven->Exsiting Maven Project选项，最后选择本示例程序所在目录即可。

使用JUnit运行VerySimpleLeaveProcessTest之后得到的结果是：

```
pid=5, pdid=leavesayhello:1:4
```

pid即流程实例在数据中的id；pdid的值有些特殊，由一些列参数组合而成并以冒号分割，其中leavesayhello就是流程定义的key，1表示版本号，4表示流程定义在数据库中的id。

除了使用Eclipse运行测试用例外，还可以在命令行中输入mvn test进入本实例目录运行。

2.4.4 添加业务节点

前面运行了最简单的例子来说明流程执行过程（严格来说不是一个流程，因为根本没有做任何事情），下面为这个例子添加一点实际业务使其可以正常工作起来。既然是请假流程就应该知道是哪个员工请了几天假（先处理不复杂的业务信息）。

首先需要为流程添加一个用户任务（userTask）来处理申请，根据申请内容决定运行申请还是驳回申请。下面先用如图2-7所示的流程图来表示需求。

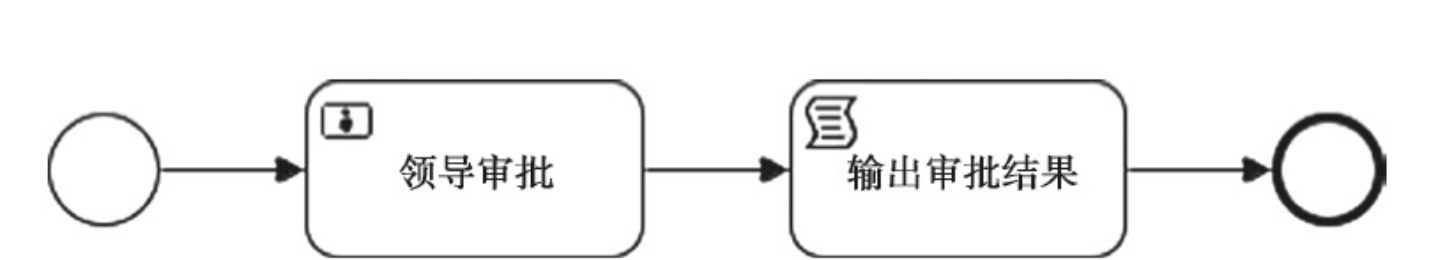


图2-7 带审批的请假流程

然后再来看看流程定义是如何设计的，以及用户任务（userTask）和脚本任务是如何定义的。支持领导审批的请假流程定义如代码清单2-4所示。

代码清单2-4 支持领导审批的请假流程定义

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:activiti="http://activiti.org/bpmn"
  xmlns:bpmndi="http://www.omg.org/spec/BPMN/20100524/DI" xmlns:omgdc="http://www.omg.org/spec/DD/20100524/DC"
  xmlns:omgdi="http://www.omg.org/spec/DD/20100524/DI" typeLanguage="http://www.w3.org/2001/XMLSchema"
  expressionLanguage="http://www.w3.org/1999/XPath" targetNamespace="http://www.kafeitu.me/activiti-in-action">
  <process id="SayHelloToLeave" name="SayHelloToLeave">
    <startEvent id="startevent1" name="Start"></startEvent>
    <userTask id="usertask1" name="领导审批" #1-S
      <potentialOwner>
        <resourceAssignmentExpression>
          <formalExpression>deptLeader</formalExpression>
        </resourceAssignmentExpression>
      </potentialOwner>
    </userTask> #1-E
    <endEvent id="endevent1" name="End"></endEvent>
    <sequenceFlow id="flow1" name="" sourceRef="startevent1"
      targetRef="usertask1"></sequenceFlow>
    <sequenceFlow id="flow2" name="" sourceRef="outputAuditResult"
      targetRef="endevent1"></sequenceFlow>
    <scriptTask id="outputAuditResult" name="输出审批结果" #2-S
      <script><![CDATA[out.println "applyUser:" + applyUser + " ,days:" + days + " ,approval:" + approved;]]></script>
    </scriptTask> #2-E
    <sequenceFlow id="flow3" name="" sourceRef="usertask1"
      targetRef="outputAuditResult"></sequenceFlow>
  </process>
</definitions>
```

此流程在2.4.1的基础上添加了新的userTask，并且增加了一个sequenceFlow定义。照例还是来解释一下此流程的定义。

#1处使用userTask来定义图2-7中的“领导审批”节点，其id为deptLeaderAudit，在其内部使用BPMN 2.0标准的用户任务分配定义元素设置了此任务由角色为deptLeader的人员处理，即有deptLeader角色的人员都可以处理此任务。

#2处定义了通过scriptTask来输出“领导审批”节点的处理结果。目前Activiti支持的scriptTask类型有Javascript和Groovy两种。本例使用可以运行在JVM上的脚本语言Groovy输出结果，语法简洁明了，读者比较容易理解。在scriptTask标签内部定义了脚本要处理的脚本内容，由script标签包裹并定义为CDATA类型数据。在流程运行的过程中Activiti会把脚本及流程变量转交给Groovy处理（需要添加Groovy的jar或依赖）。

这一步迈得好像有点大了，从一个空流程突然多出了两个不同的任务，不过这两个任务都比较简单。下面结合代码清单2-5的单元测试代码来讲解一下流程，相信读者能很快了解流程的运行过程以及任务的作用。

代码清单2-5 SayHelloToLeaveTest.java

```
public class SayHelloToLeaveTest {
    @Test
    public void testStartProcess() throws Exception {
        ProcessEngine processEngine = ProcessEngineConfiguration
            .createStandaloneInMemProcessEngineConfiguration().buildProcessEngine();
        RepositoryService repositoryService = processEngine.getRepositoryService();
        String bpmnFileName = "me/kafeitu/activiti/helloworld/SayHelloToLeave.bpmn"; #1-S
        repositoryService.createDeployment().addInputStream("SayHelloToLeave.bpmn ", this.getClass().getClassLoader()
            .getResourceAsStream(bpmnFileName)).deploy(); #1-E
        ProcessDefinition processDefinition = repositoryService
            .createProcessDefinitionQuery().singleResult();
        assertEquals("SayHelloToLeave", processDefinition.getKey());
        RuntimeService runtimeService = processEngine.getRuntimeService(); #2-S
        Map<String, Object> variables = new HashMap<String, Object>();
        variables.put("applyUser", "employeeel");
        variables.put("days", 3);
        ProcessInstance processInstance = runtimeService.startProcessInstanceByKey(
            "SayHelloToLeave", variables); #2-E
        assertNotNull(processInstance);
        System.out.println("pid=" + processInstance.getId() + ", ppid="
            + processInstance.getProcessDefinitionId());
        TaskService taskService = processEngine.getTaskService(); #3-S
        Task taskOfDeptLeader = taskService.createTaskQuery()
            .taskCandidateGroup("deptLeader").singleResult();
        assertNotNull(taskOfDeptLeader);
        assertEquals("领导审批", taskOfDeptLeader.getName()); #3-E
        taskService.claim(taskOfDeptLeader.getId(), "leaderUser"); #4
        variables = new HashMap<String, Object>(); #5-S
        variables.put("approved", true);
        taskService.complete(taskOfDeptLeader.getId(), variables); #5-E
        taskOfDeptLeader = taskService.createTaskQuery() #6-S
            .taskCandidateGroup("deptLeader").singleResult();
        assertNull(taskOfDeptLeader); #6-E
        HistoryService historyService = processEngine.getHistoryService(); #7
        long count = historyService.createHistoricProcessInstanceQuery().finished() #8
            .count();
        assertEquals(1, count);
    }
}
```

为了更好地理解代码，在开始讲解此代码清单之前先以情景模拟的方式来了解任务的签收与办理过程：公司的每个部门可能有多个领导（A是正手，B是副手），在将一项任务分配给“部门领导”角色之后所有的部门领导都会收到一条未签收状态的任务。假如公司约定请假流程的审核由B处理，在申请人申请请假之后A和B同时会看到一个需要处理的任务（Task），B签收任务并办理，此时A的待办任务列表中就少了一项任务。本来一项任务是属于一个角色或某几个候选人的，在执行了签收动作之后任务归签收人所有。在了解对任务的签收和办理的执行过程之后我们再来看看代码清单2-5的执行过程。

#1处和代码清单2-3的#3-E处一样用来部署流程，只不过变换了一种方式，不是直接部署bpmn20.xml而是部署以bpmn为扩展名的文件。原因解释一下：这个流程定义是通过Eclipse Designer创建的，默认扩展名为bpmn（实际内容是一样的），在Activiti 5.9以及之前的版本中流程引擎在读取流程定义文件时必须以bpmn20.xml结尾，所以这里需要变通一下，在部署流程的时候读取一个文件流并由foo.bpmn20.xml传入，结果和代码清单2-3中部署bpmn20.xml一样。从Activiti 5.10版本开始Activiti已经支持直接部署以bpmn扩展名结尾的流程定义文件。

#2处与代码清单2-3中的启动方式稍微有些差别，在调用启动流程的方法中传入了一个Map集合变量，这里设置了两个属性，applyUser表示申请人的名称，days表示请假的天数。这样Activiti在启动流程的时候会把这两个变量存入数据库中，以后就可以通过接口读取到节点。

#3处的任务是查询组（Group）deptLeader的未签收任务并验证任务的名称。

#4处通过taskService调用claim方法“签收”此任务归用户leaderUser所有。

#5处就是领导的处理结果。在流程启动时填写了请假人与请假天数来模拟实际场景，领导审批通过，也就是在#5-S处设置变量approval为true表示审批通过，在#5-E处完成任务的同时以流程变量的形式设置审批结果。

#6处是为了让读者更好地理解执行结果，因为任务已经办理完成，再次查询组deptLeader的任务已经为空。

#7处通过流程引擎对象获取历史记录查询接口。

#8处通过历史接口统计已经完成（finished）的流程实例数量，接着验证预期的结果。

细心的读者可能会问：明明流程图中有两个节点（领导审核和输出审批结果），为什么在代码中没有处理scriptTask呢？在讨论这个问题之前先看看运行结果：

```
pid=5, pdid=SayHelloToLeave:1:4
applyUser:employee1 ,days:3, approval:true
```

结果中第一行和代码清单2-3执行结果相同，第二行则不同。第二行的输出是scriptTask执行的结果，在流程定义中用Groovy脚本输出使用变量拼接的信息。现在可以来回答刚刚提出的问题：为什么代码没有处理scriptTask呢？原因很简单，对于scriptTask，流程引擎会自动处理，处理完成之后流转到下个节点，在本例中scriptTask之后就是结束事件了，流程结束，所以没有处理scriptTask。

就目前来说，除了userTask不能被自动处理之外，其他的任务均由流程引擎自动处理，无需人工参与。

[1] 笔者还专门买了一件印有Hello World的T恤。

2.5 Activiti Explorer

前面提到了Activiti Explorer是Activiti为了让开发人员快速入门所设计的一个示例程序，本节将介绍如何运行Activiti Explorer以及如何部署和处理任务。

2.5.1 配置并运行Activiti Explorer

如果读者下载的是最新版本，那么直接把activiti-5.1x（5.11及之后的版本）的wars/activiti-explorer.war复制到一个干净的Tomcat的webapps目录后运行Tomcat即可。如果需要运行5.10及之前的版本，需要根据下面的步骤依次操作。不管运行哪个版本，最后启动应用的访问路径看到如图2-8所示的界面。

现在我们重新回到Activiti解压缩的目录，进入setup目录，例如笔者的Activiti解压目录是：/home/henryyan/work/sources/activiti/activiti-5.10/setup。此目录中有几个子目录和配置文件，下面依次介绍。

- **files**：此目录中包含运行Activiti Explorer所需要的一些配置文件及Web应用，例如，Tomcat配置、数据库（H2）初始化数据、所需的jar包等。
- **build.db.properties**：用来配置数据库信息，通过配置db属性可以使用其他的数据库，例如，MySQL、Oracle、SQL Server等。此文件中还有jdbc的配置信息，可以通过更改这几项配置来使用本地数据库。
- **build.properties**：用过Ant的读者对此肯定很熟悉了，需要把Ant运行时的配置信息单独配置到一个文件。此文件中配置了运行时使用的Tomcat版本及自动文件存放位置。
- **build.xml**：配置了运行、停止、清理等目标。

运行Activiti Explorer比较简单，只要使用Ant执行事先定义好的目标即可。现在通过命令进入安装目录（activiti-5.x/setup），执行命令ant demo.start即可自动构建、运行目标，并且在完成之后自动打开浏览器，访问地址为：<http://localhost:8080/activiti-explorer>，打开的页面如图2-8所示。

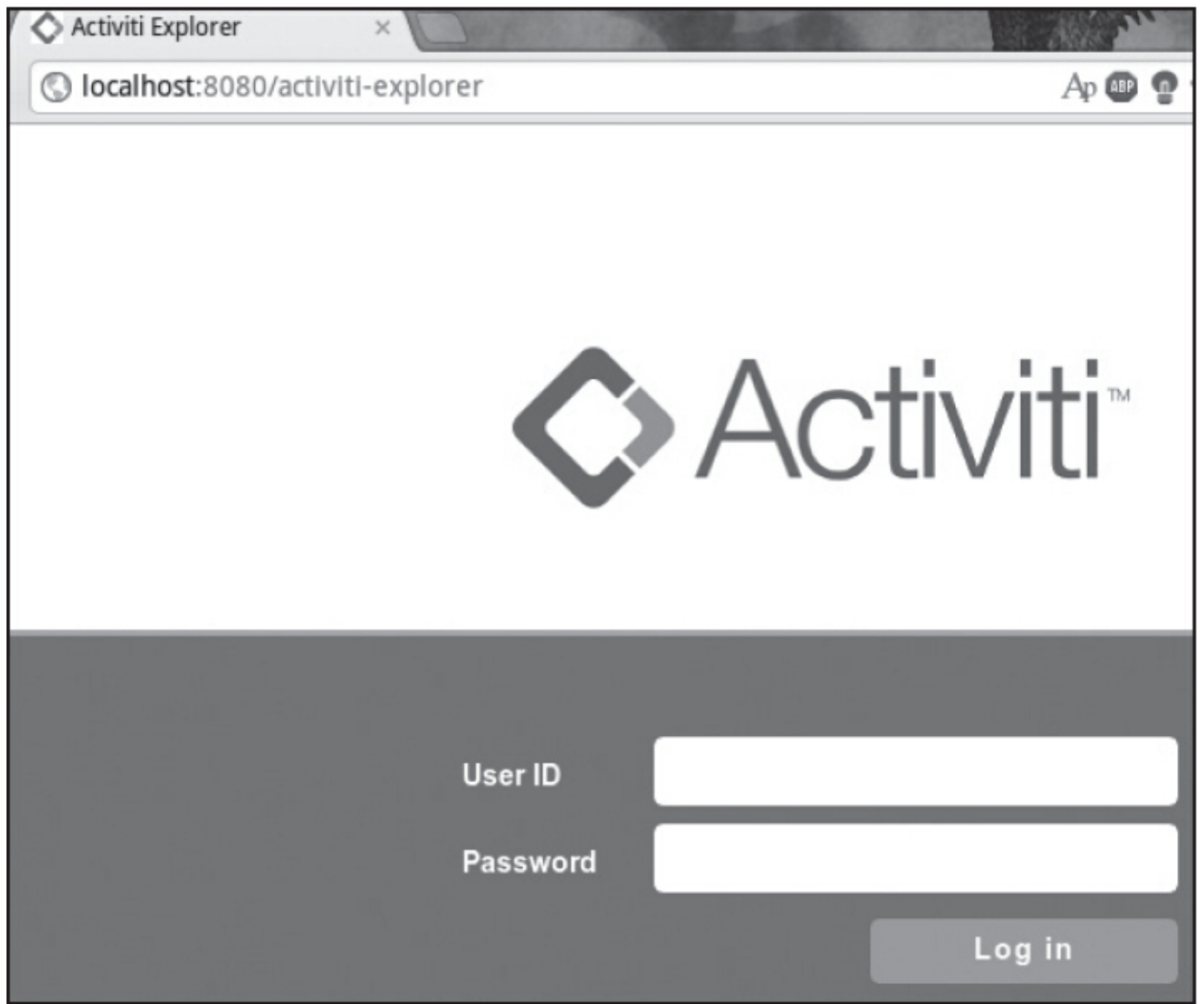


图2-8 Activiti Explorer登录页面截图

2.5.2 使用Activiti Explorer

1.登录系统

在通过Ant脚本启动Activiti Explorer的过程中已经自动初始化了用户和组数据，这些数据文件位于activiti-5.x/setup/files/demo/h2.data.sql；对于5.11及之后的版本在启动Activiti Explorer的时候系统会自动执行数据初始化工具类插入初始化数据。

以拥有管理员角色的用户kermit登录系统，默认密码与用户名kermit相同。登录之后的页面如图2-9所示。

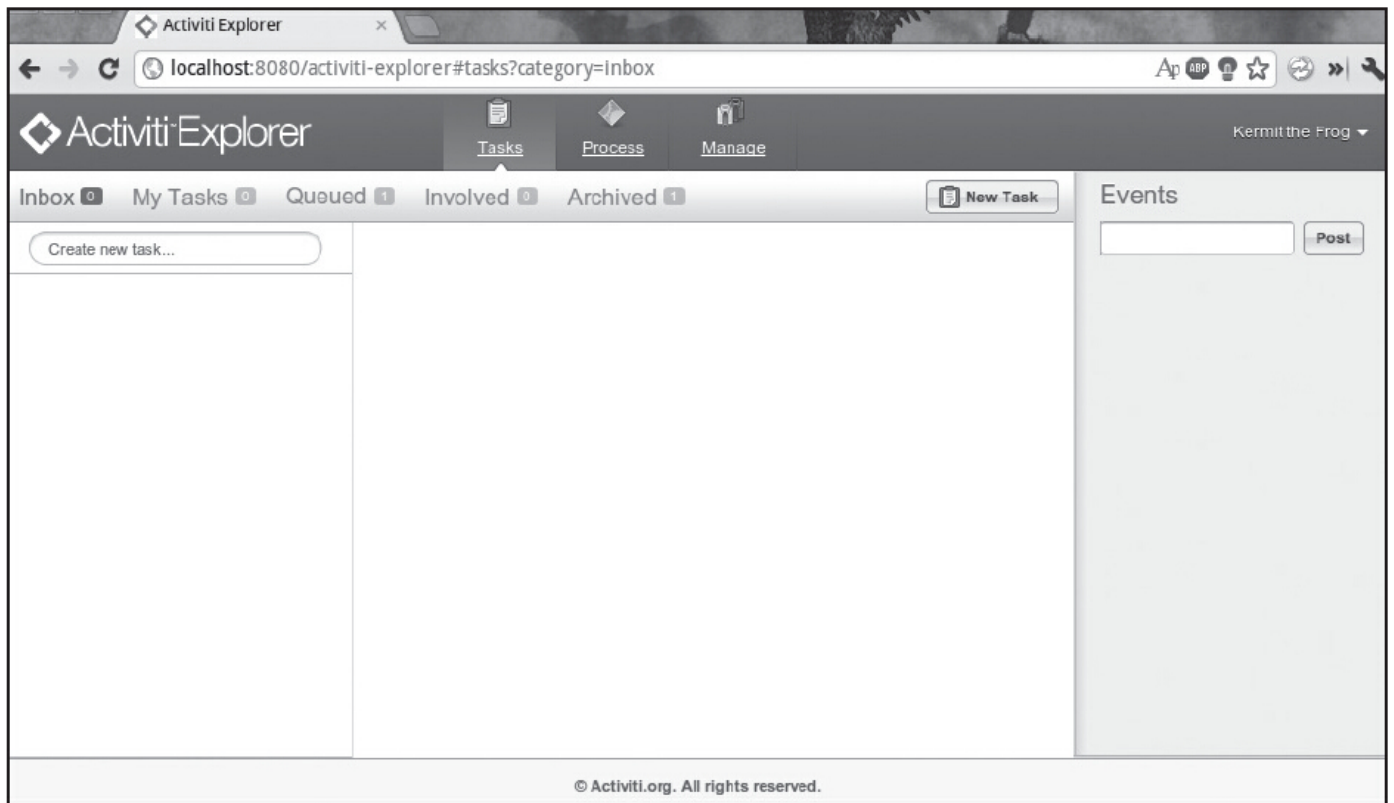


图2-9 登录Activiti Explorer之后的页面截图

2.部署流程

单击“Manage”栏目，然后单击“Deployments”菜单选择“Upload new”，弹出的对话框如图2-10所示。

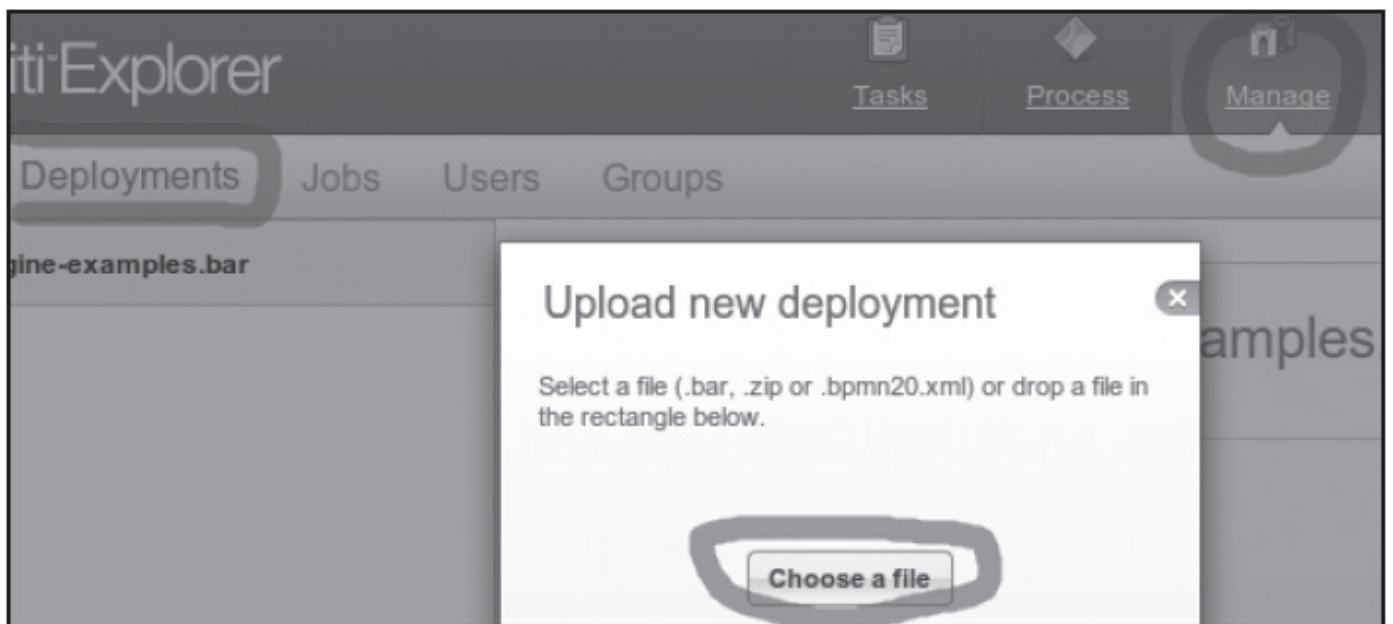


图2-10 在Activiti Explorer中部署流程

前面设计的流程SayHelloToLeave需要在启动流程时设置变量，而在Activiti Explorer中为了简单演示笔者采用表单（form）形式单独设计了流程定义SayHelloToLeaveForActivitiExplorer。单击“Choose a file”，然后在文件选择对话框中选择bpmn20-example/src/main/resources/me/kafeitu/activiti/helloworld/SayHelloToLeaveForActivitiExplorer.bpmn20.xml，这样Activiti Explorer会自动部署流程并跳转到流程资源查看页面。

部署流程之后单击“Process”栏目，在左侧单击“SayHelloToLeaveForActivitiExplorer”可以查看流程的图片形式，此图片由引擎自动根据xml的配置信息生成。但是很不幸我们又遇到了让人郁闷的乱码问题，如图2-11所示。

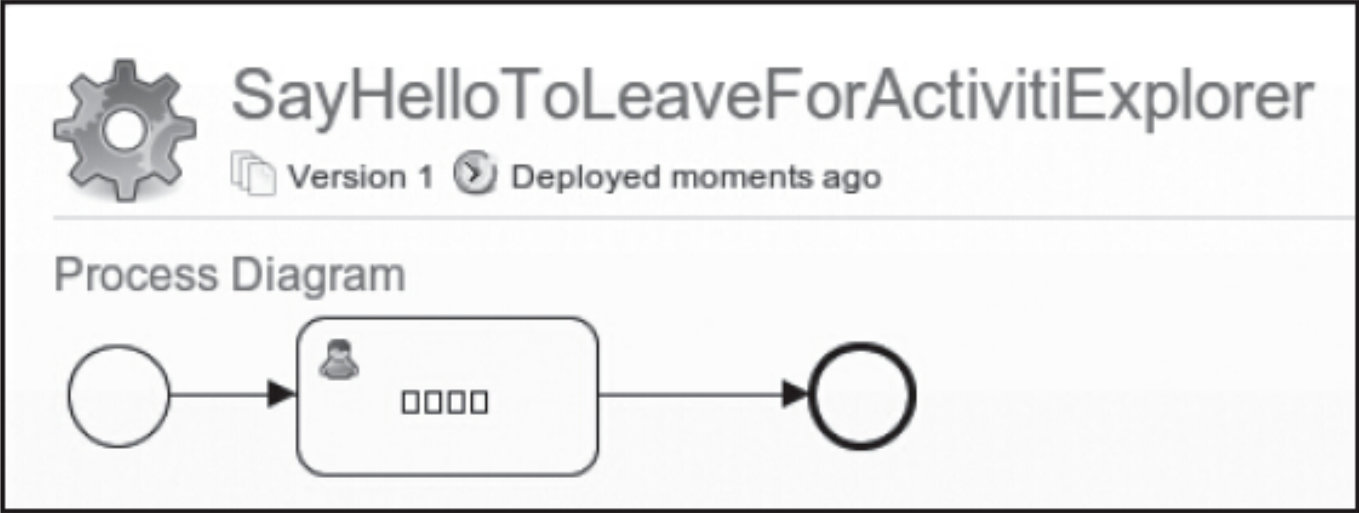


图2-11 在Activiti Explorer中部署流程后的中文乱码

具体解决办法会在5.3.4节中详细介绍。现在可以再次部署，但是这次选择SayHelloToLeaveForActivitiExplorer.zip压缩包，这个压缩包中有两个文件：SayHelloToLeaveForActivitiExplorer.bpmn20.xml和SayHelloToLeaveForActivitiExplorer.png。

重新部署之后可以正常显示中文，如图2-12所示。注意划线处的Version 2，它表示同一个流程定义部署了两次，版本号自动累加。



图2-12 部署SayHelloToLeaveForActivitiExplorer.zip之后中文正常显示

3.启动流程

在“Process”栏目中单击左侧的“SayHelloToLeaveForActivitiExplore”，然后单击页面右上角的“Start process”，跳转到启动流程表单页面，如图2-13所示。



SayHelloToLeaveForActivitiExplorer



Version 2



Deployed 10 minutes ago

Apply User*

days*

Start process

Cancel

图2-13 启动SayHelloToLeaveForActivitiExplorer流程页面

填写Apply User和days之后单击“Start process”按钮即可启动流程。

4.签收与办理任务

在此流程中，节点“领导审批”被设置为Management组。当前登录用户Kermit拥有此组，所以在启动此组之后单击“Task”栏目可以看到“Queued”后面显示数字1，表示当前有需要处理的任务，如图2-14所示。

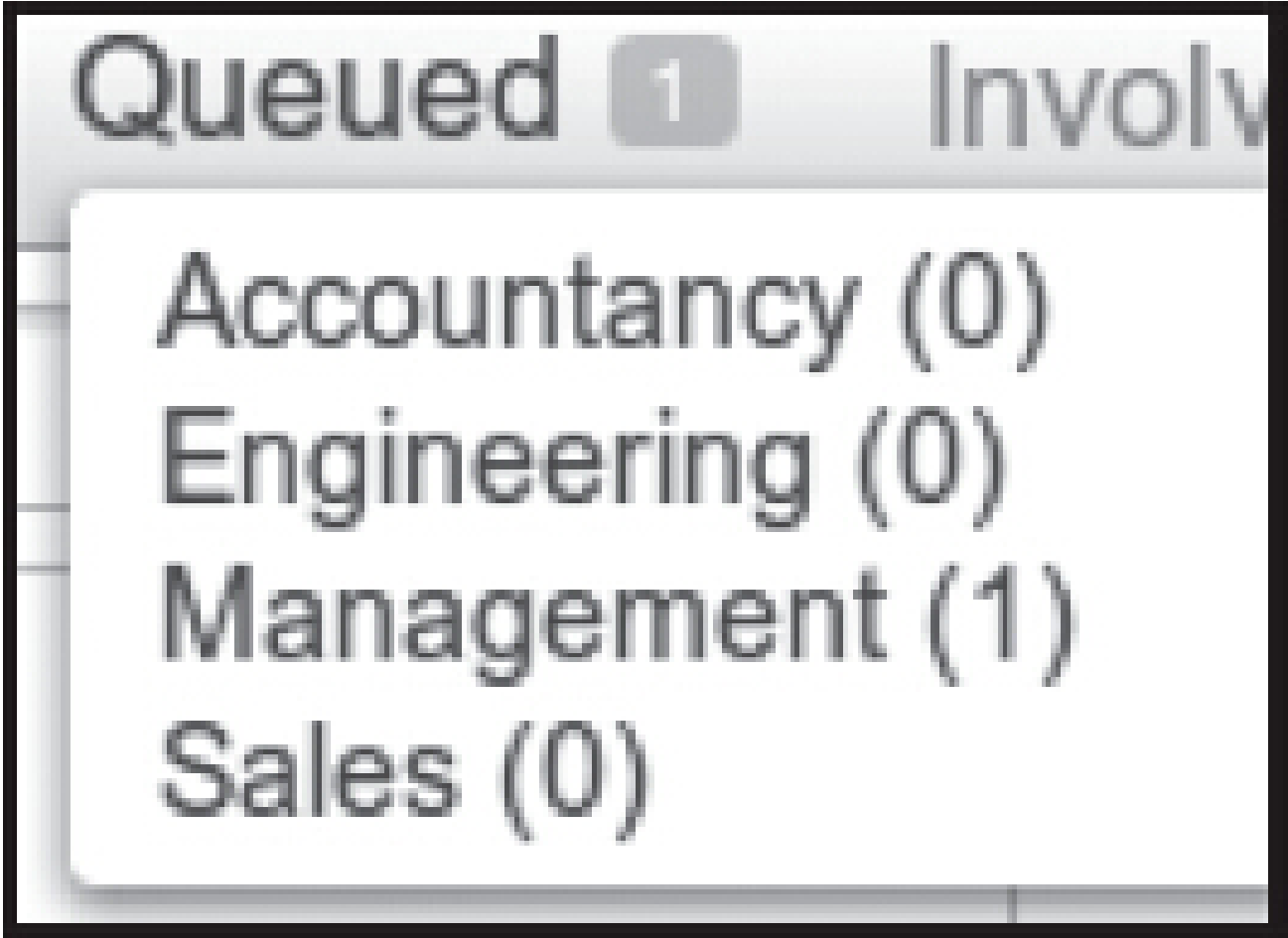


图2-14 Management组的待处理任务

单击“Management(1)”之后显示任务办理页面，其中包含任务的名称、任务所属人及表单信息，如图2-15所示。



领导审批

 No due date

 Medium Priority

 Created 4 minutes ago

Claim

This case has no description set.

点击即可签收此任务

Part of process: 'SayHelloToLeaveForActivitiExplorer'

People

 No owner

Transfer

 No assignee

Reassign

Subtasks

No subtasks defined for this task

Related content

No related content attached for this task

Fill in the form below and complete the task:

Apply User

张三

days

2

Approval *

Complete task

Reset form

图2-15 任务办理对话框

我们已经了解了签收与办理的过程了，现在单击“Claim”按钮之后任务归kermit所属，之后就可以审批请假请求了。在“Approval”下拉框中选择一项审批结果，如图2-16所示。

Related content

No related content attached for this task

Fill in the form below and complete the task:

Apply User

张三

days

2

Approval *

同意

Complete task

Reset form

图2-16 选择审批结果

选择审批结果后单击“Complete task”即完成了任务的办理。此流程比较简单，仅一个要点是学会使用Activiti Explorer。

2.6 本章小结

本章从下载Activiti开始介绍，接着对配置JDK、Ant、Maven环境进行讲解，然后在此基础上逐步介绍最简单的Hello World程序，紧接着又进阶到如何使用userTask及scriptTask，并且利用单元测试验证所的期望结果，最后就如何运行Activiti Explorer进行讨论，使读者学会如何部署流程并解决中文乱码问题，以及如何在Activiti Explorer上启动、签收、完成任务。相信读者现在已经对Activiti开发环境有了一个大体地了解，本章接触到的启动事件、结束事件、用户任务、脚本任务只是BPMN 2.0规范中很小的一部分，下一章我们将学习Activiti目前支持的BPMN 2.0规范。

第二部分 基础篇

在第一部分中用两章内容介绍了Activiti的背景知识和Activiti入门，从整体上对Activiti有了初步的了解，讲述了Activiti、工作流的历史，之后通过简单的例子讲解如何使用Activiti API驱动流程。本部分内容都是后续章节的基础。

第3章涉及两种流程设计工具Activiti Modeler和Activiti Designer，分别适用于业务人员和开发人员，详细介绍了这两种工具如何安装以及使用，尤其是利用Activiti Designer和JUnit单元测试帮助读者在Activiti引擎基础上更好地理解BPMN 2.0规范。

第4章的Activiti与BPMN 2.0规范可以说是全书的理论基础，讲述了BPMN 2.0规范中模型的图形、XML描述和Activiti在其基础上的扩展。有兴趣的读者可以结合BPMN 2.0规范的官方文档^[1]辅助学习。

[1] <http://www.org/spec/BPMN/2.0/PDF/>

第3章 流程设计工具

前面两章我们首先从概念上介绍了工作流和Activiti的背景知识，然后在第2章中通过日常办公中请假的例子打开了Activiti世界的大门。在第2章的例子中提到的流程定义文件是XML格式的，这样的文件

全部手工来写肯定是不现实的，本章将带领读者学习如何使用工具拖拽的方式设计流程。

Activiti针对不同的平台提供了不同架构的流程设计器：基于B/S架构的流程设计器Activiti Modeler（可以通过浏览器在线设计流程），基于Eclipse插件的流程设计器Activiti Designer。

3.1 基于B/S架构的流程设计器Activiti Modeler

Activiti Modeler是基于B/S架构的流程设计器，在Activiti 5.11版本发布之前Activiti官方提供了独立的Activiti Modeler组件，需要开发人员执行压缩包中的脚本打包，但是从Activiti 5.11开始不需要自行打包Activiti Modeler了，在下载压缩包中已经把重新设计的Activiti Modeler集成到了Activiti Explorer中，所以不想了解如何打包的读者可以跳过本节内容。关于重新设计的Activiti Modeler将在3.2节介绍。

3.1.1 Activiti Modeler特点

在介绍Activiti Modeler之前不得不先介绍一下Signavio，它是Activiti Modeler背后的Signavio开源BPMN设计器。Activiti Modeler并非Activiti官方开发的，而是基于Signavio^[1]公司开发的Signavio构建的。

Signavio是一款成熟BPMN可视化在线设计器，底层基于德国的HPI公司开发的Web的建模工具Oryx^[2]。Signavio虽然是一家商业公司，但是它提供了遵循MIT协议开源的版本signavio-core-components。

Signavio公司同时提供了商业版本的设计器以满足更多、更强的需求，比如团队协作、高级流程设计器、流程Portal、分析与报表、导入/导出不同格式的文件（ARIS、XPDL、PDF、Visio、Excel）。当然用户在正式购买商业版本之前有30天的试用期来了解商业版本的功能。

Activiti Modeler完全使用signavio-core-components的源码构建，而且在打包时仅需要简单的配置即可。

 **说明** 在Activiti 5.6及之前的版本中，Activiti Modeler是众多组件中的一部分，不需要我们自己打包war文件，有兴趣的读者可以下载Activiti 5.6版本的Zip包解压一下看看。

3.1.2 下载signavio-core-components

signavio-core-components的源码托管在Google Code上，项目地址：<http://code.google.com/p/signavio-core-components/>。先利用版本控制器（Subversion）检出源码到本地工作区。有Subversion客户端的读者可以利用客户端方便地检出源码，只需要在URL中输入<http://signavio-core-components.googlecode.com/svn/trunk/>即可。喜欢使用命令行的读者可以执行以下命令：

```
svn checkout http://signavio-core-components.googlecode.com/svn/trunk/
```

检出之后的文件列表如图3-1所示。

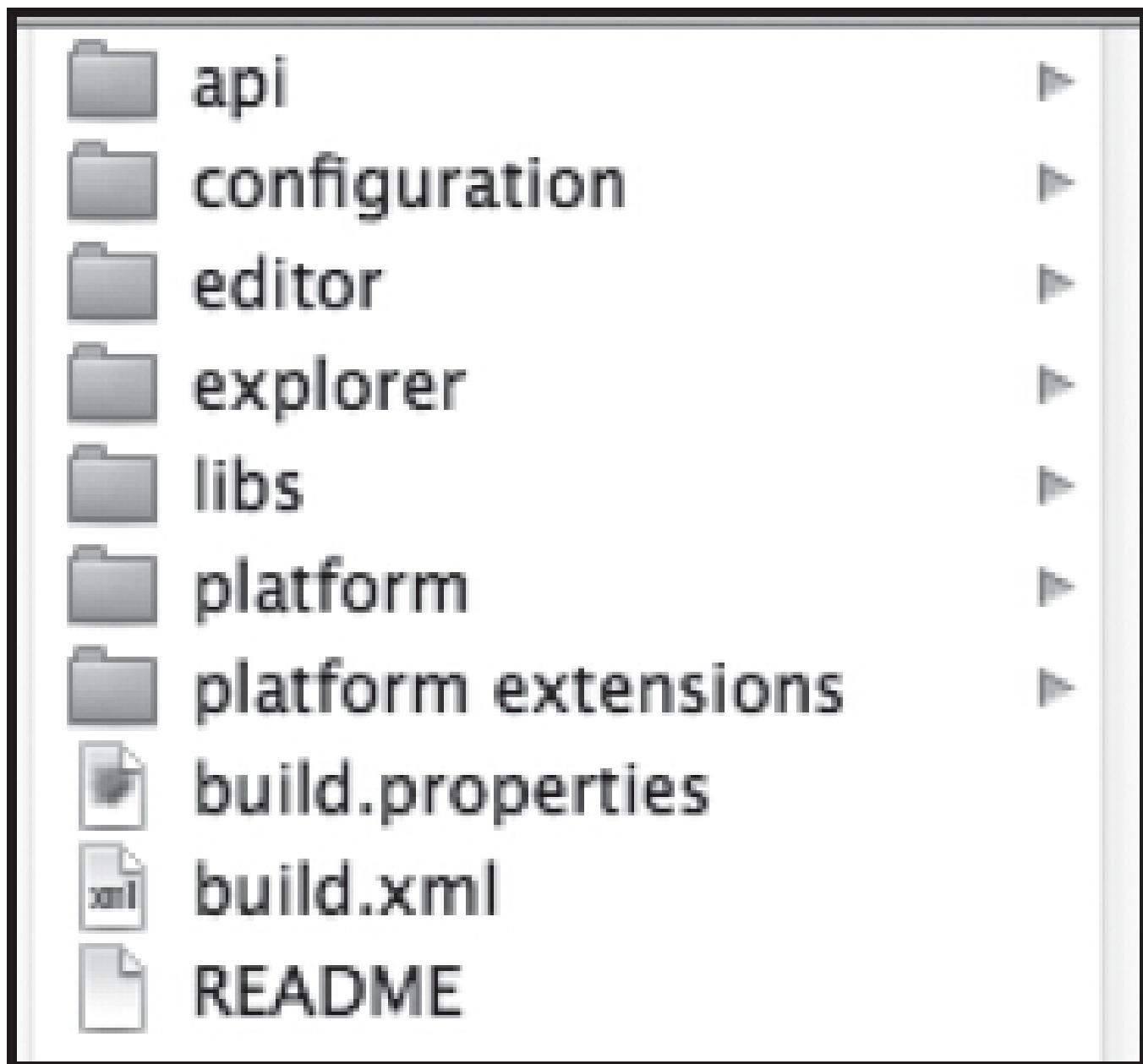


图3-1 signavio-corecomponents目录结构

对于目录中的文件，我们只关心build.properties和build.xml即可。如果读者对其他目录文件有兴趣可以逐个浏览，以便日后在此基础上进行扩展。

3.1.3 配置打包与运行

build.properties文件就是打包时需要配置的一些选项的属性文件。用编辑器将该文件打开之后看到的文件内容如图3-2所示。

下面介绍配置文件中的属性及其含义。

- `dir-tomcat-webapps`: 运行打包结果（war格式）的tomcat路径。
- `dir-jboss-webapps`: 运行打包结果（war格式）的jboss路径。
- `target`: 存放构建结果的目录，是打包之后生成的war文件的存放位置。
- `version`: 设计器的版本号，可完全由用户自定义。
- `war`: 打包的war文件的名称，例如activiti-modeler，打包之后会在target目录下生成activiti-modeler.war文件。
- `configuration`: 使用的配置，包含一些CSS样式和JSON格式的编辑器配置。目前支持3种配置：default、Activiti、jBPM。有兴趣的读者可以进入到configuration目录一探究竟。
- `host`: 打包之后运行时的路径。这个配置很关键，还要注意仅仅需要提供服务器地址（域名或IP）及端口即可。
- `fileSystemRootDirectory`: signavio运行时的工作区目录，会保存编辑器的配置及设计的流程定义文件。值得注意的是，不管在何种操作系统环境中都只使用“/”作为路径分隔符。


```
build.properties  ×
1  # This is the Signavio Core Components configuration file. Most users only have to configure this file.
2
3  # The path to your Apache Tomcat webapps folder
4  dir-tomcat-webapps = /apache-tomcat-6.0.16/webapps
5
6  # The path to your jBoss deployment folder
7  dir-jboss-webapps = /jboss-5.1.0/server/default/deploy
8
9  # The folder the war file(s) is/are stored
10 target = target
11
12 # The version of the application. If you want to integrate the Signavio Core Components into
13 # your own software product, you can align the version number.
14 version=1.0.0
15
16 # The name of the war file, if you use the all-in-one-war build target
17 war = signaviocore
18
19 # The configuration you want to use. This is the name of the folder in the 'configuration' project
20 # that contains the configuration and skin files. The following configurations are available:
21 # default, Activiti, jBPM
22 # You can also add your own configuration in the 'configuration' project.
23 configuration = default
24
25 # The URL of your server. Format: http(s)://<domain>(:<port>)
26 # Do not add a trailing slash here!
27 host = http://localhost:8080
28
29 # The path on your system the directories and diagram files are created.
30 # Do not use \ ! Always use / !
31 fileSystemRootDirectory = C:/repo
```

图3-2 build.properties默认配置

了解了配置之后读者就可以根据自己的实际环境更改配置了。笔者只更改了几个属性，其他属性均使用默认值。

```
version=5.9
war = activiti-modeler
configuration = Activiti
fileSystemRootDirectory=/Users/henryyan/workspaces/activiti-modeler
```

了解到这里读者完全可以基于signavio-core-components构建自己的流程设计器，并且在此基础上进行改造，例如整合公司的其他产品、功能扩展以及本地化等。

更改好配置之后只需要执行`ant build-all-in-one-war`命令来执行打包任务，执行结果如下：

```
-henryyan@hy-mbp ~/work/sources/signavio-core-components$ ant build-all-in-one-war
Buildfile: /Users/henryyan/work/sources/signavio-core-components/build.xml
...
[war] Building war:
/Users/henryyan/work/sources/signavio-core-components/target/activiti-modeler.war
BUILD SUCCESSFUL
Total time: 27 seconds
```

执行完以上命令后我们可以看到在signavio-core-components/target目录中生成了`activiti-modeler.war`文件，如图3-3所示。

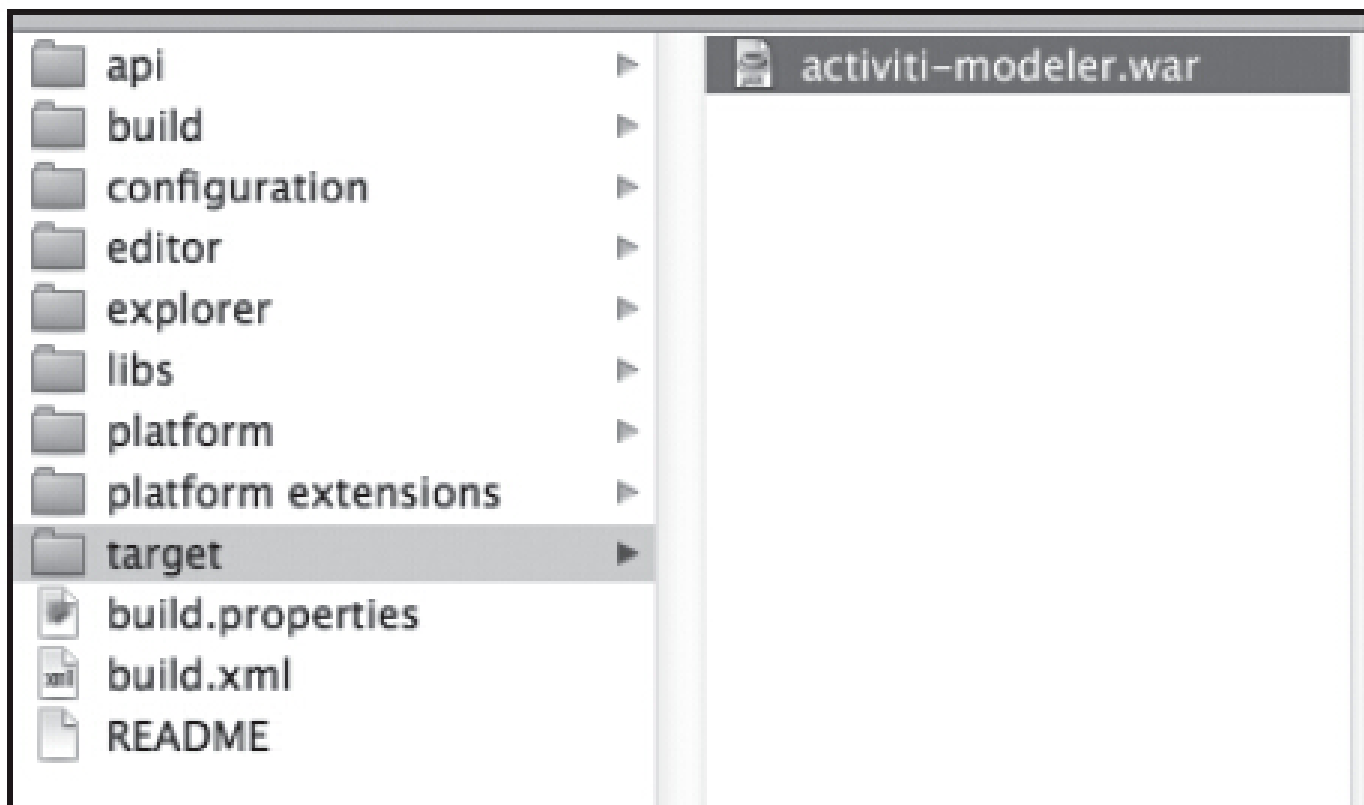


图3-3 打包命令的执行结果

打包之后复制target/activity-modeler.war到本地Tomcat的webapps目录，然后启动Tomcat，之后在浏览器中输入<http://localhost:8080/activiti-modeler>访问应用，加载页面如图3-4所示。

 **提示** 目前Signavio仅支持非IE内核的浏览器，例如Chrome、Firefox、Safari。



图3-4 Activiti Modeler的加载页面

从图3-4中可以看出，图片使用的是configuration/Activiti中的signavio_logo.png，版本号就是刚刚在build.properties中定义的version属性。

在加载完成之后进入Activiti Modeler的首页，页面如图3-5所示。

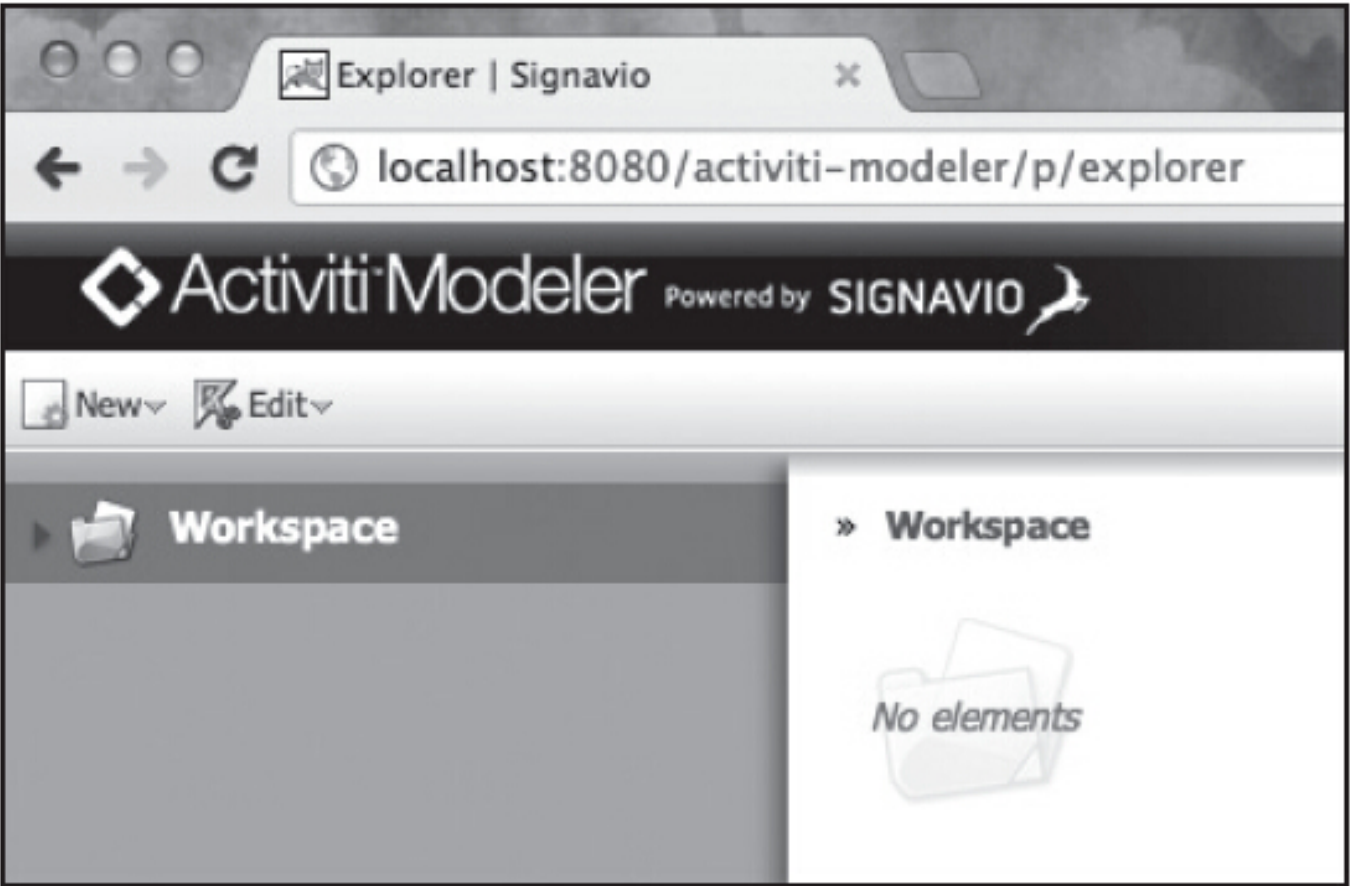


图3-5 Activiti Modeler 首页

3.1.4 Windows用户打包

本书是针对Windows用户的老话题——乱码。

Windows用户在执行ant的打包命令`ant budil-all-in-one-war`时会遇到如图3-6所示的错误信息。

```
BUILD FAILED
C:\work\signavio-core-components\build.xml:64: The following error occurred while
executing this line:
C:\work\signavio-core-components\editor\build.xml:119: Java returned: 2

Total time: 2 seconds
```

图3-6 Windows环境下打包Activiti Modeler报错

导致构建错误的原因是编码不一致，Signavio的源码采用的是UTF-8编码，而中文版本的Windows操作系统默认的编码为GBK，编码不一致导致在转换的时候报错。笔者在最初学习Activiti时使用的是Linux操作系统（Ubuntu^[3]），在撰写本书的时候又转向了Mac OS X系统，因为Linux/UNIX默认的编码是UTF-8，所以没有遇到图3-6中的错误。

解决方法分三步，在用编辑器打开signavio-core-components/editor/build.xml文件进行以下处理。

1) 找到<target name= “com.signavio.editor.js.concat” >，紧随其后添加一行配置代码：

```
<property name="charset" value="utf-8"/>
```

将标签中的<concat destfile= ‘\${build}/oryx.debug.js’ >修改为：

```
<concatdestfile=${build}/oryx.debug.js' encoding="${charset}" outputencoding= "${charset}">
```

最终的配置如下：

```
<target name="com.signavio.editor.js.concat">
<property name="charset" value="utf-8"/>
<concat destfile=${build}/oryx.debug.js' encoding="${charset}" outputencoding= "${charset}">
```

2) 找到<target name='com.signavio.editor.js.compress'>代码，更改此target内的<java dir='\${build}' jar='\${root}/lib/yuicompressor-2.4.2.jar' fork='true' failonerror='true' output='\${compress.temp}'>，将其中的yuicompressor-2.4.2.jar更改为yuicompressor-2.4.7.jar。

3) signavio默认使用yuicompressor-2.4.2版本压缩javascript和css文件。为了解决编码问题我们需要使用最新的yuicompressor版本来替换2.4.2版本。在笔者撰稿时最新的yuicompressor版本为

2.4.7, 读者也可以直接下载最新版本。访问<http://yuilibrary.com/download/yuicompressor/>下载第一个版本的压缩包, 将其解压后提取build/yuicompressor-2.4.7.jar文件并复制到signavio-core-components/yuicompressor/editor/lib目录中。

再次执行打包命令`ant build-all-in-one-war`不会提示如图3-6所示的错误提示, 如图3-7所示。

```
[copy] Copying 2 files to C:\work\signavio-core-components\platform\target\classes
[war] Building war: C:\work\signavio-core-components\target\activiti-modeler.war

BUILD SUCCESSFUL
Total time: 28 seconds
```

图3-7 解决了Windows环境下打包Activiti Modeler错误后的提示

3.1.5 设计请假流程

首先单击图3-5中的“New”按钮, 然后选择“Business Process Diagram(BPMN 2.0)”新建一个流程, 此时浏览器会打开一个新的如图3-8中的设计器页面。

整个设计器都是基于Ext的拖曳方式进行操作, 如图片3-8所示。其中左边栏是模型仓库, 中间部分是工作区, 右边栏可以用于设置中间工作区选中模型的属性, 如果工作区中没有选中的模型, 那么右边栏可以用于设置流程主信息。

在第2章中已经初步运行了请假流程, 在这里我们仍然以请假流程为例来讲解业务需求人员如何使用Activiti Modeler从业务角度设计流程。Activiti Modeler设计的请假流程如图3-9所示。

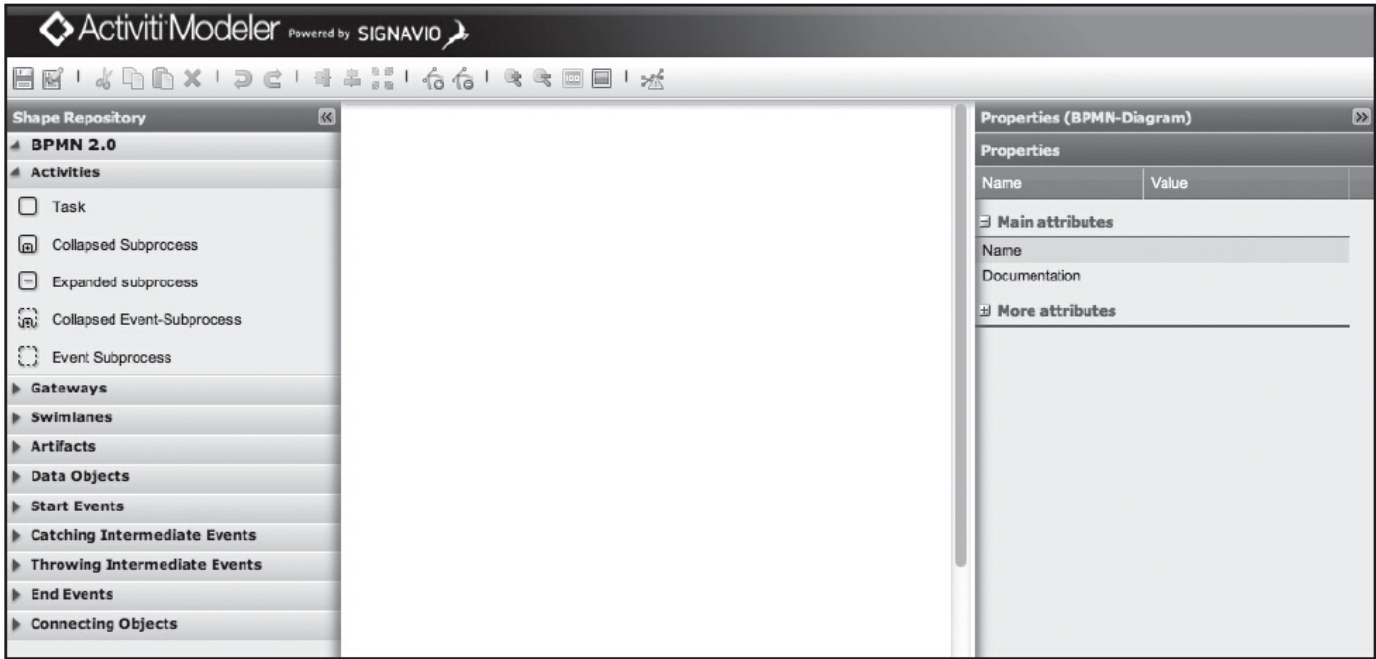


图3-8 Activiti Modeler的新建流程页面



图3-9 Activiti Modeler设计的请假流程

一个完整的流程总是以开始事件 (Start Event) 为入口, 在Activiti Modeler中只要从左侧的模型仓库找到“Start Events”, 然后用鼠标将其拖曳到中间的空白工作区中。开始事件和快捷菜单如图3-10所示。

当鼠标移动到圆形表示的“Start Event”时会出现一些快捷菜单图标, 单击这些小图标可以快速创建各种模型并自动用顺序流箭头连接两个模型。

接下来创建“领导审批”节点时可以单击空白的矩形图标快速创建一个“Task (任务)” ; 也可以从左边栏的“Activities”中拖曳“Task”到空白区域, 然后把鼠标移动到“Start Event”上单击“箭头”图标将其拖曳到刚刚创建的Task模型上。在日常的使用中还是第一种方式方便得多, 第二种方式只有在特殊情况下才会用到。

了解了如何创建Task之后就可以使用第一种方式创建“领导审批”节点了。图3-11展示了通过快速方式创建的Task。

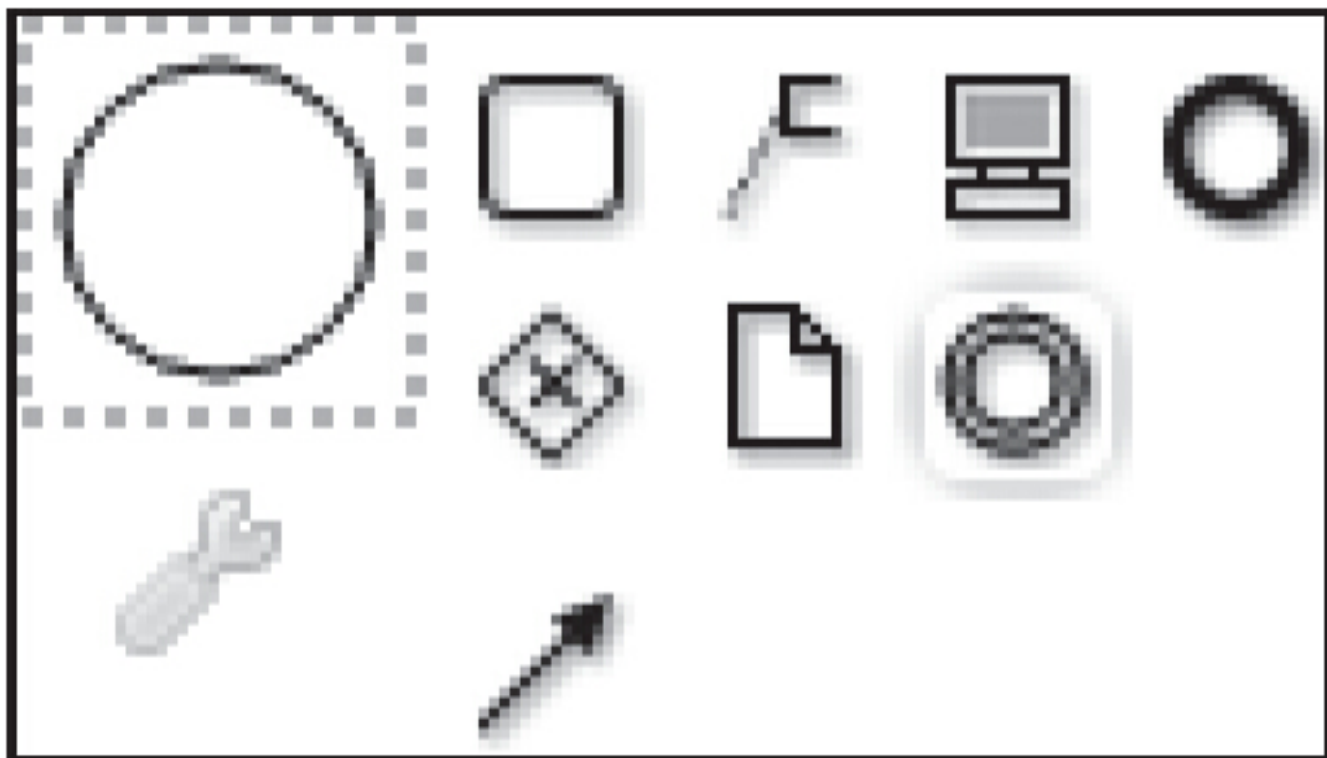


图3-10 开始事件和快捷菜单

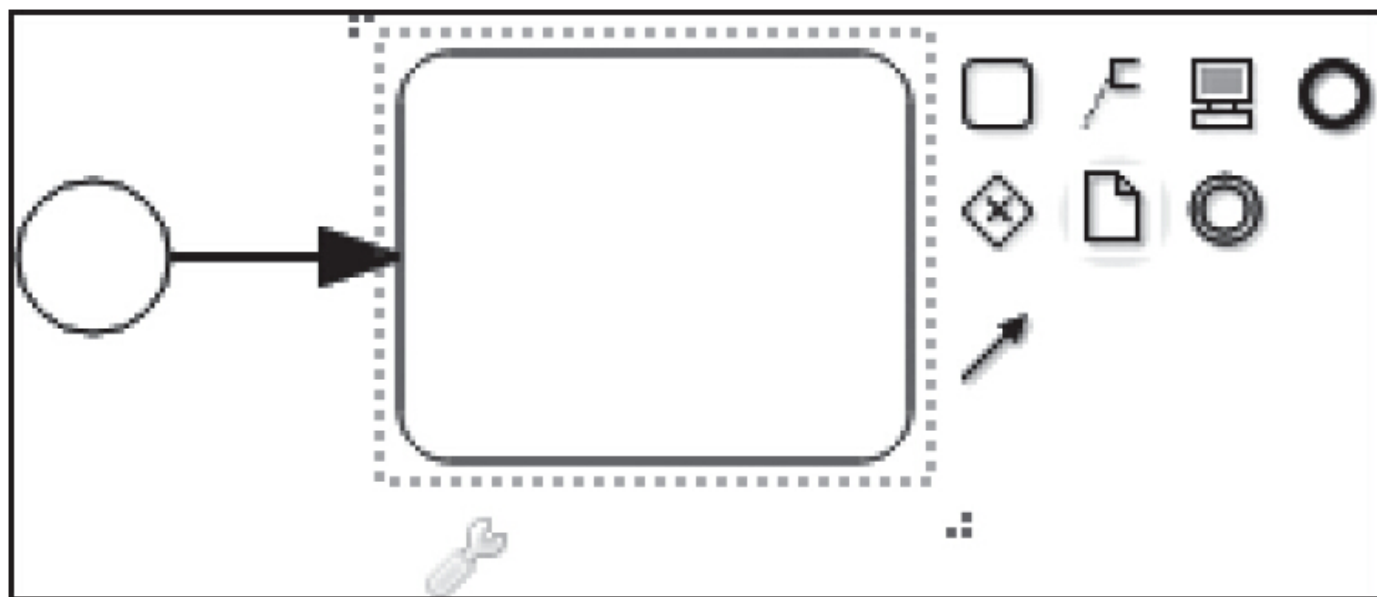


图3-11 通过快速方式创建的Task

有了Task模型之后就要进一步完善Task的属性，例如任务名称、类型等。设置属性需要展开右边栏中的“Attributes”，然后单击刚刚创建的空白Task，在右边栏中设置Task的相关属性即可。在图3-12中设置了Task的“Name”和“TaskType”。读者可以继续添加其他剩余的节点并设置其属性。

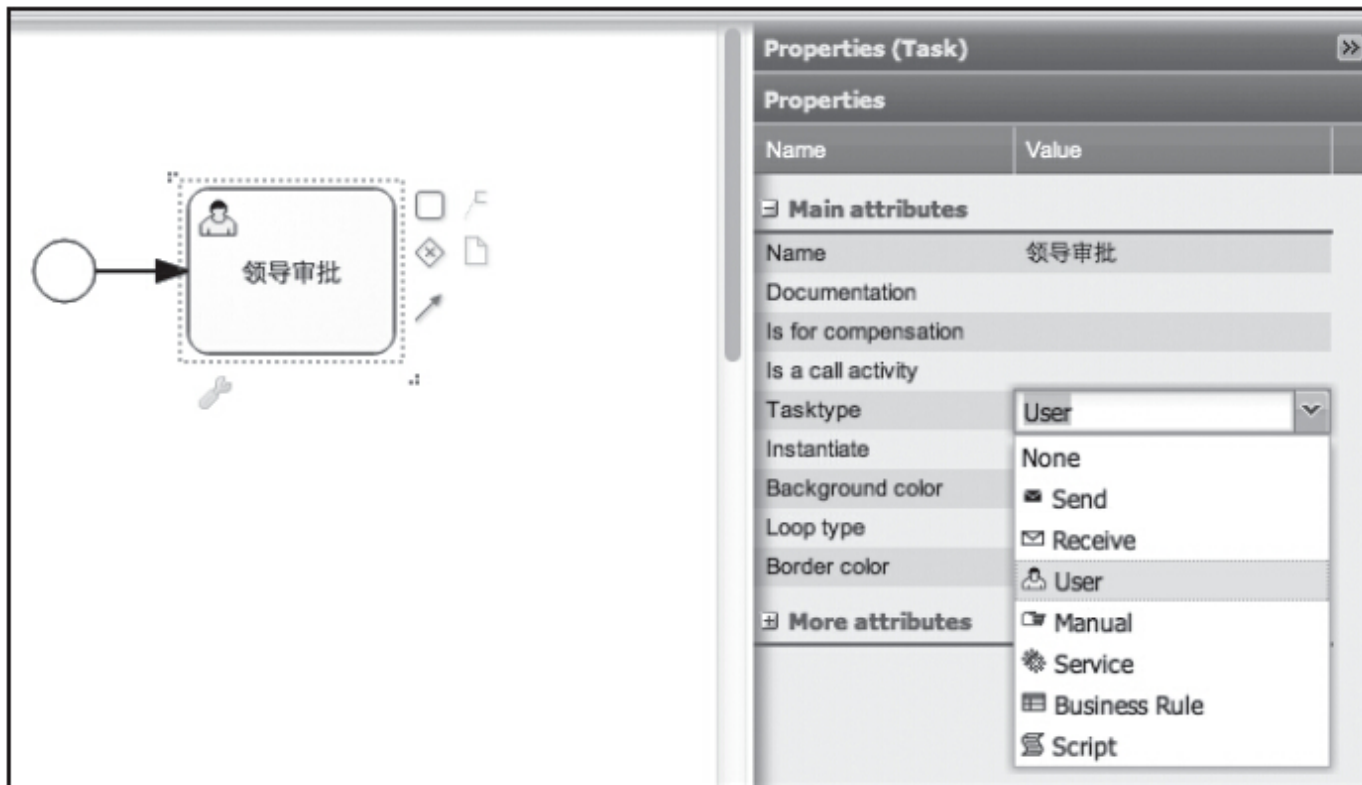


图3-12 设置领导审批节点的属性

一个完整的流程还要有结束事件（End Event），在普通的Task之后需要有一个结束节点。结束节点的创建方式和普通Task的创建方式一样，将鼠标移动到需要添加结束事件的任务上，然后单击粗线条的圆形图标就会自动创建一个结束事件，如图3-13所示。关于各种事件及模型会在第4章详细介绍。

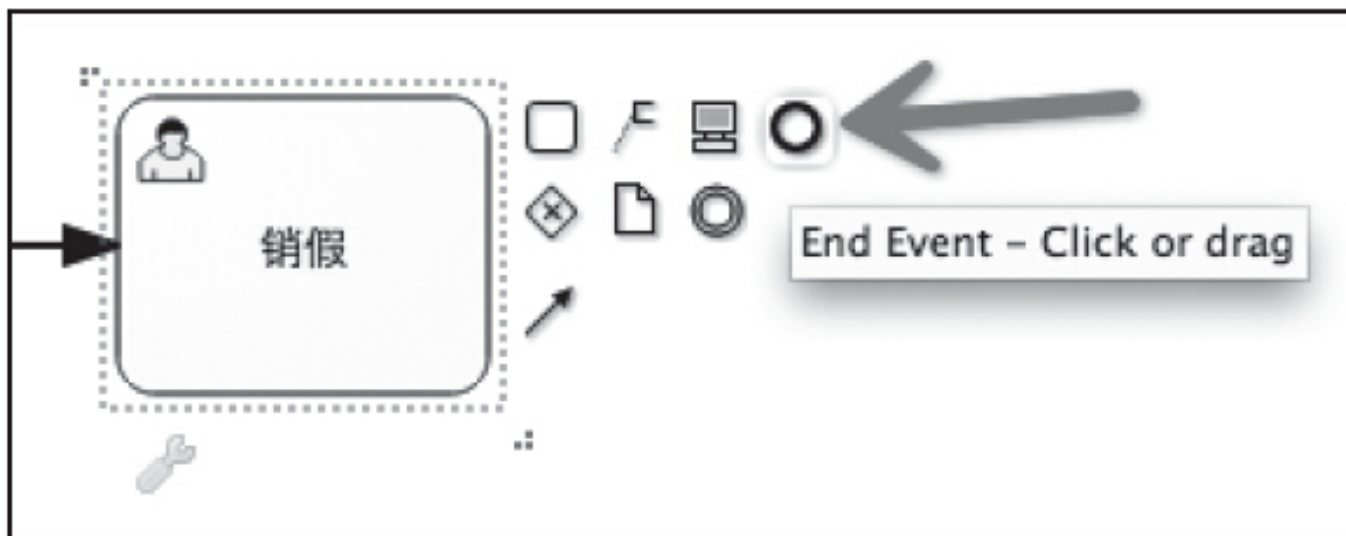


图3-13 创建结束事件（End Event）

到此一个完整的流程就设计完成了，在结束之前不要忘记保存劳动成果。单击页面左上角的保存按钮，弹出如图3-14所示的对话框。

 **说明** 在填写“Title”字段时不要输入中文，否则会导致再次打开流程图时系统报错，提示找不到页面（404错误），报错信息如图3-15所示。或许以后的版本能够解决这个问题。

保存流程之后可以在Activiti Modeler主页面的Workspace中看到流程定义，选中流程定义可以进行编辑、移动、复制、删除等操作，如图3-16所示。

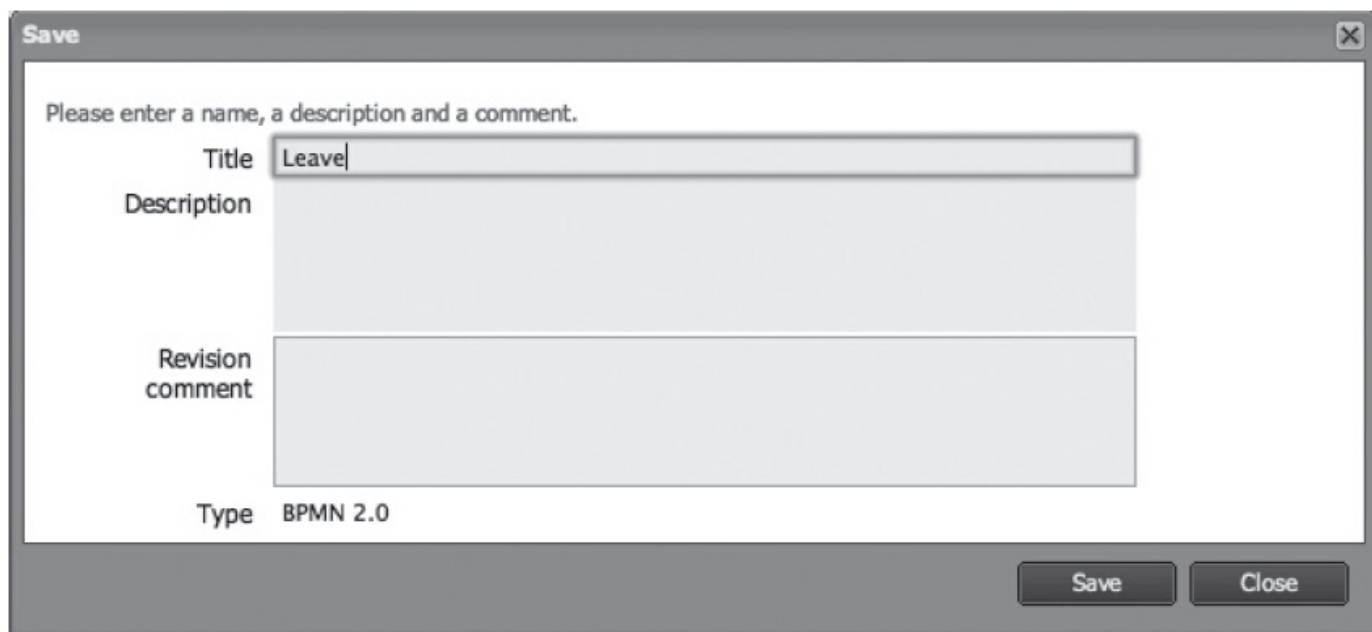


图3-14 保存流程对话框

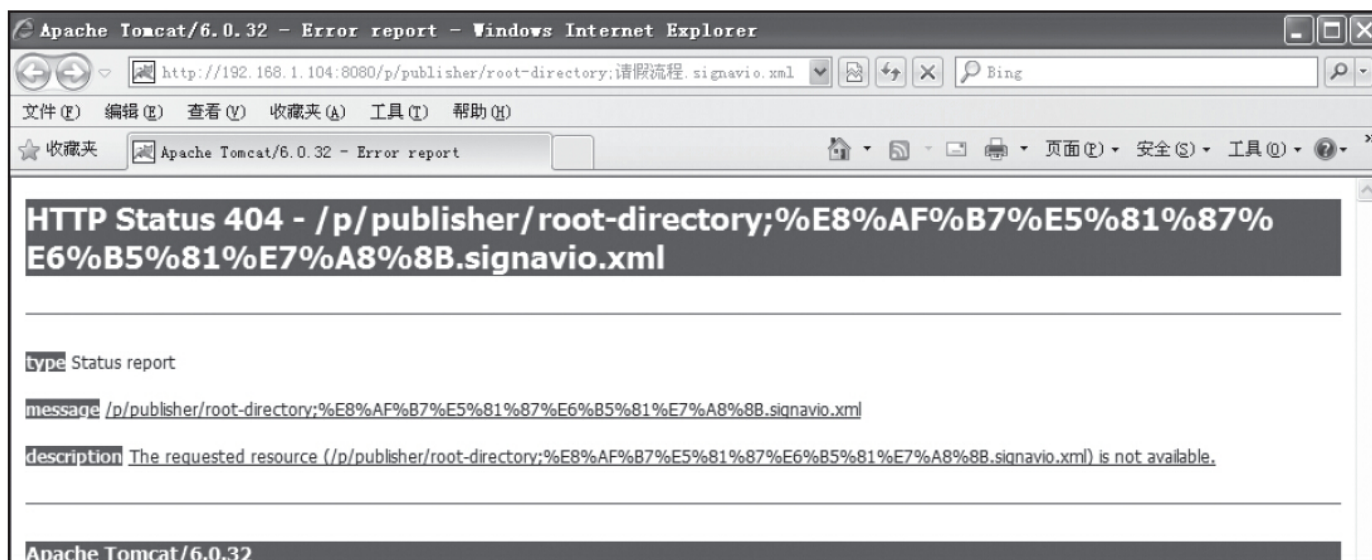


图3-15 打开中文名称的流程图导致404错误

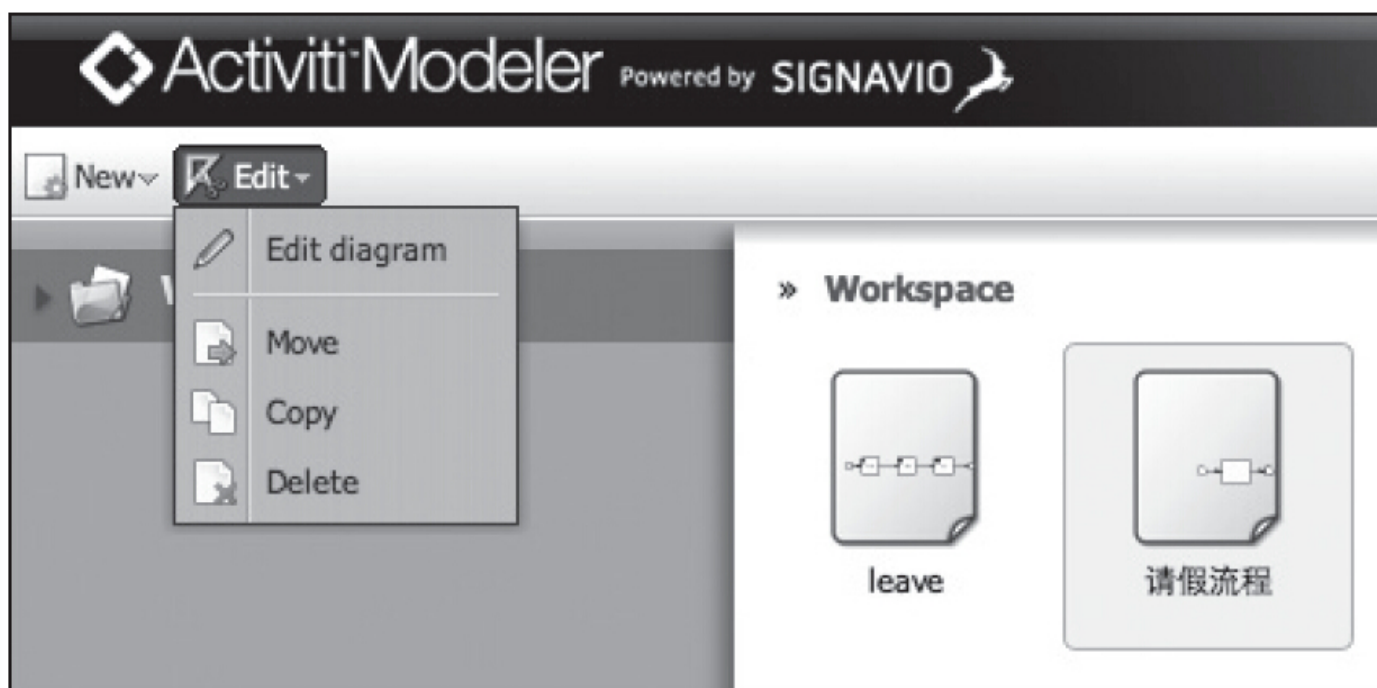


图3-16 Activiti Modeler主页面的Workspace

3.1.6 导出bpmn2.0.xml

图3-16中有两个流程定义，一个是英文名称的，一个是中文名称的。Activiti Modeler的设计结果都以xml的形式保存在磁盘上，保存xml文件的目录就是在最初打包时设置的fileSystemRootDirectory属性值。例如笔者的设置为/Users/henryyan/work/workspaces/activiti-modeler，打开该目录可以看到共有4个文件，如图3-17所示。



图3-17 流程设计文件列表

从图3-17中可以看出，每个流程定义都有两个文件，只不过扩展名不同。其中“bpmn20.xml”为流程定义文件，我们在第2章中部署流程的时候使用的也是扩展名为bpmn20.xml的xml文件；“signavio.xml”文件用于保存设计流程所需要的一些配置，其中配置以JSON方式保存在leaven.signavio.xml文件的“json-representation”属性中。有兴趣的读者可以打开流程文件阅读其中的配置，本书不深入讨论。

[1] <http://www.signavio.com/>
[2] <http://code.google.com/p/oryx-editor/>
[3] 一个很流行的Linux 发行版本，可从<http://www.ubuntu.com>上下载。

3.2 在Activiti Explorer中使用Activiti Modeler

从5.11版本开始官方提供的压缩包发生了较大变化（参考2.1.1节）：把重新设计的Activiti Modeler整合到了Activiti Explorer中，可以直接创建新模型然后部署到引擎中，也可以根据已有的流程定义创建模型，修改后可以直接把最新的修改部署到引擎。

提示 由于Activiti Modeler组件需要依赖REST服务，因此读者要了解如何把Activiti Modeler集成到自己的项目中，相关内容可参考20.8节。

Activiti 5.11版本之后的Activiti Modeler依然在Signavio的基础上开发，由KISBPM^[1]提供开源版本，同时KISBPM也提供商业版本支持更多的功能，例如模型的版本控制、表单设计器、自定义属性、模板库、模型部署、角色控制等。

新版Modeler移除了原Signavio对于其他规范的支持（例如BPM），只保留Activiti支持的BPMN 2.0规范，并且支持Activiti扩展的活动以及属性。

把压缩包中提供的activiti-explorer.war部署到Tomcat中即可使用Activiti Modeler。使用kermit/kermit登录后单击“Processes”→“Model workspace”，即可看到如图3-18所示的页面（根据系统的语言不同界面显示的语言也不同，笔者在5.11版本发布前为Activiti Explorer添加了国际化的中文部分）。



图3-18 模型工作区

单击“New Model”按钮后打开创建模型对话框，如图3-19所示。

New model

New model

Name *

请假流程

Description

Cancel

Create

图3-19 创建新模型对话框

新版设计器的界面和老版本一致（图3-8），如图3-20所示。

Activiti Modeler powered by KIS BPM

Shape Repository

BPMN 2.0

Start Events

Activities

Structural

Gateways

Boundary Events

Intermediate Catching Events

Intermediate Throwing Events

End Events

Connecting Objects

Artifacts

模型库

工作区

Properties (BPMN-Diagram)

Properties

Name

Value

Main attributes

Name

Documentation

Process identifier process

Execution listeners

More attributes

属性区域

图3-20 新版Activiti Modeler界面

从左侧选择模型后拖动到工作区即可，单击模型可以在右侧设置该模型的属性，如图3-21所示。

从图3-21中右侧部分可以看到有表单属性（Form properties）、监听器（Task listeners）等设置。

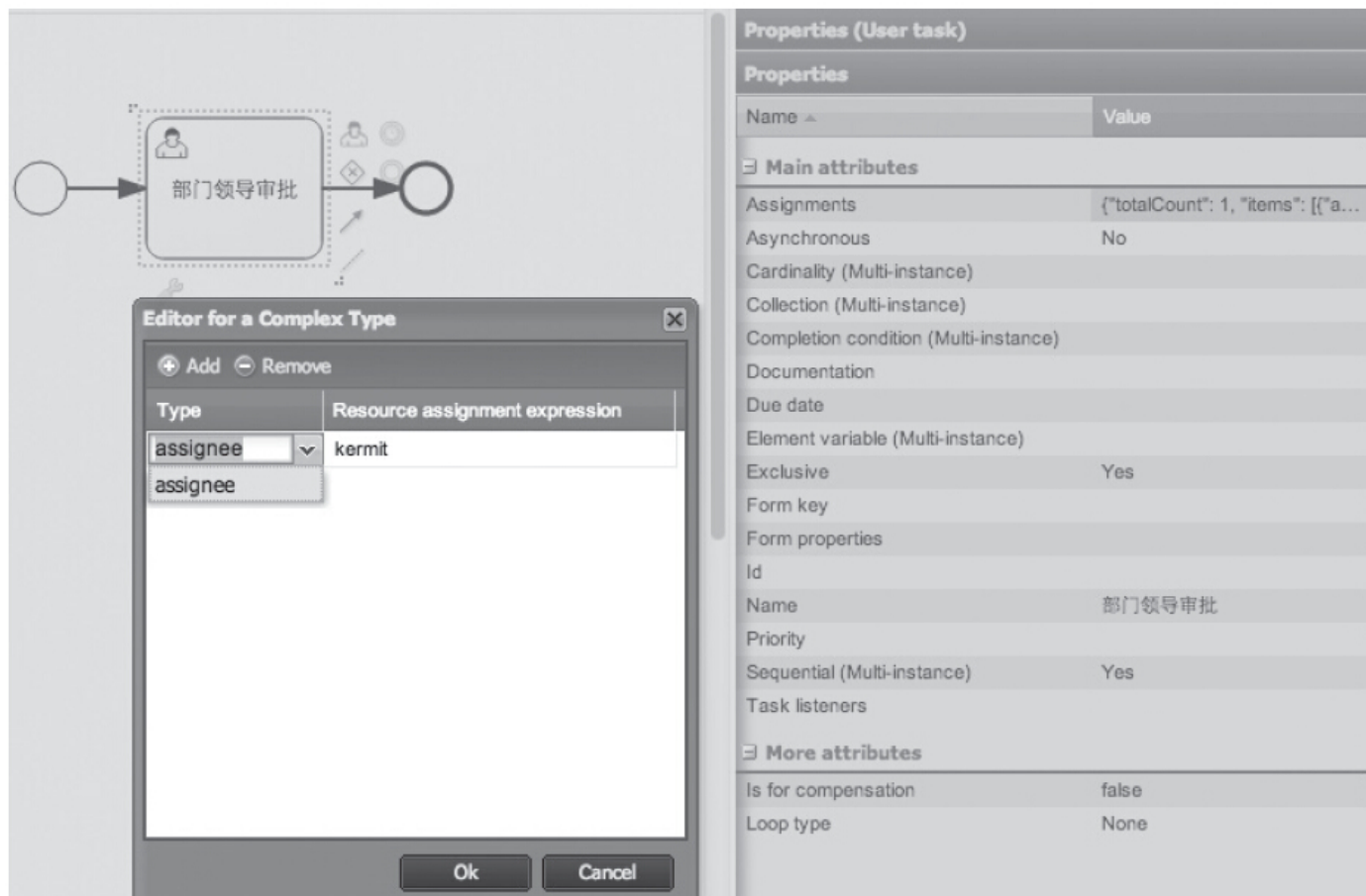


图3-21 设置模型的属性

编辑完模型之后单击图3-20左上角的保存按钮保存模型，再单击右上角的“X”关闭设计器返回到Activiti Explorer中，如图3-22所示。

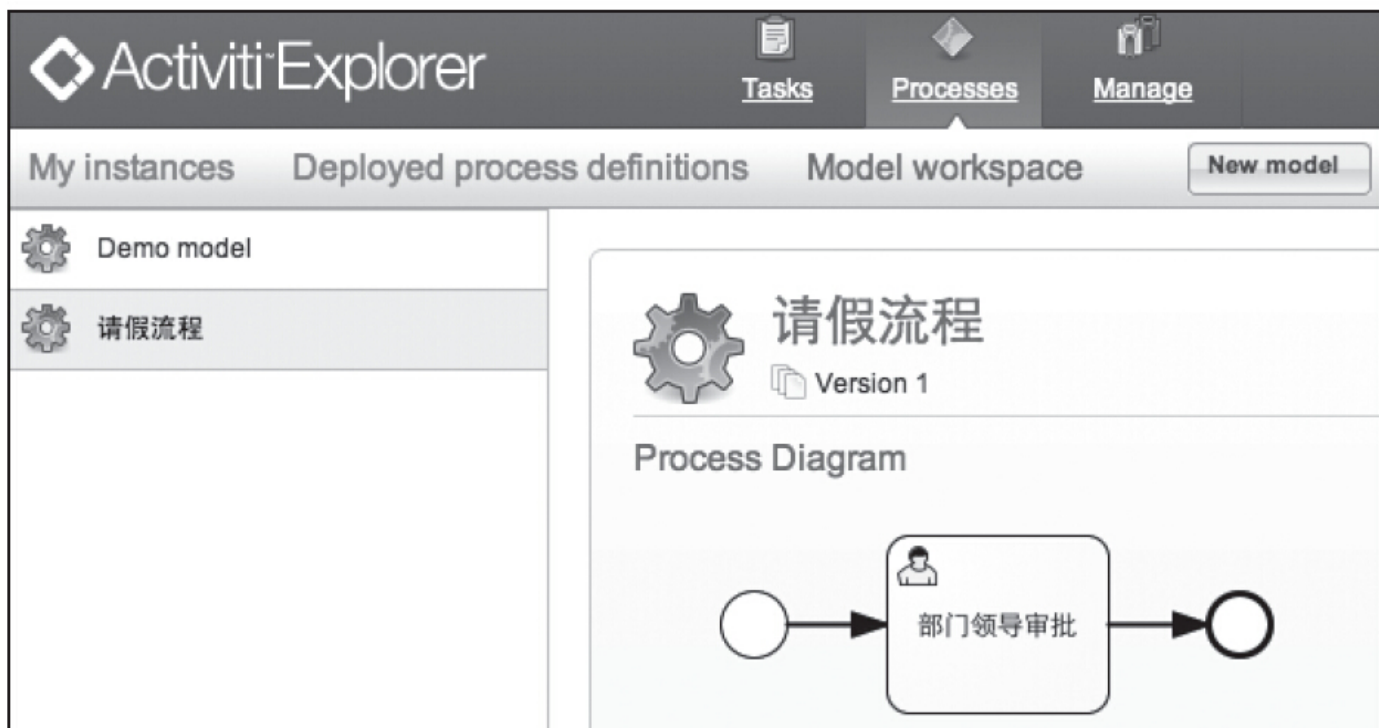


图3-22 保存模型后返回到Activiti Explorer界面显示设计的请假流程

单击图3-18中右上角的“Deploy”即可部署该模型，在流程定义界面就可以看到刚刚部署的流程定义，如图3-23所示。



图3-23 从模型工作区部署的流程定义

[1] <http://www.kisbpm.com/>

3.3 基于Eclipse插件的流程设计器Activiti Designer

在客户的需求确定之后，由业务人员利用Activiti Modeler设计完业务流程，此时可以将设计结果导出为bpmn20.xml文件，之后由开发人员继续基于设计进一步添加涉及技术细节的配置，例如排他分支的条件、Java服务、任务监听器等。

3.3.1 Activiti Designer特点

Java程序员每天工作使用的主要工具是IDE，而应用最广泛的IDE当然是Eclipse，任何开发人员都可以很容易地开发Eclipse的插件。Activiti目前提供了Eclipse的Activiti Designer插件，以后还会陆续推出其他IDE的插件，例如NetBeans、IntelliJ IDEA等。

Activiti Designer一般随着Activiti引擎一起发布，在Activiti引擎的新版本添加了对BPMN 2.0规范的支持和实现之后，Activiti Designer也同步更新了支持新规范的可视化流程设计。

相对于Activiti Modeler严格以BPMN 2.0规范为基础实现可视化设计，Activiti Designer提供了Activiti自行扩展的十几种配置（以后会陆续介绍），例如使用activiti:assignee简化UserTask的任务办理人属性，以及使用activiti:candidateGroup简化任务候选组等。

TDD（Test Driven Development，测试驱动开发）被广泛用于日常开发中。Activiti Designer当然也为TDD提供了良好的支持：开发人员可以基于流程定义文件（foo.bpmn）快速生成测试用例，在设计流程的时候把存在“坏味道”的业务流程停止。

3.3.2 安装Activiti Designer

本书使用的Eclipse版本为4.2（代号Juno），Activiti Designer支持3.7（代号Indigo）以后的Eclipse版本。

建议读者采用Site方式安装Activiti Designer：单击菜单“Help” -> “Install New Software”，打开如图3-24所示的对话框。然后单击“Add”按钮，在打开的对话框中填写以下内容，接着单击“OK”按钮确认添加Repository。

- Name: Activiti BPMN 2.0designer
- Location: <http://activiti.org/designer/update/>

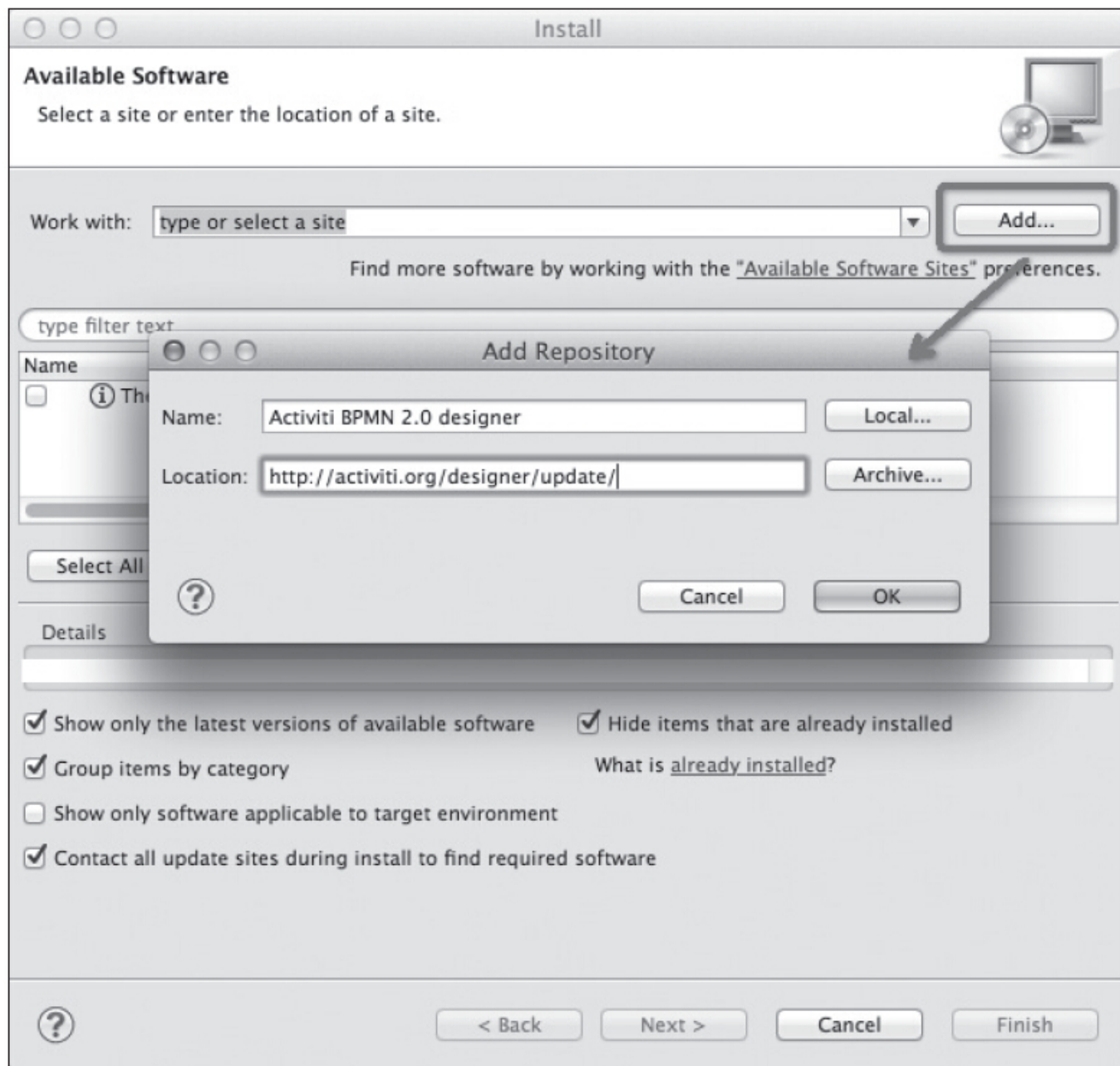


图 3-24 用Site方式添加Activiti Designer的Repository

3.3.3 设计流程

1.新建Activiti项目

单击“File”→“New”，选择图3-25中的“Activiti Project”项目，单击“Next”进行一些设置之后就可以创建一个Activiti项目。

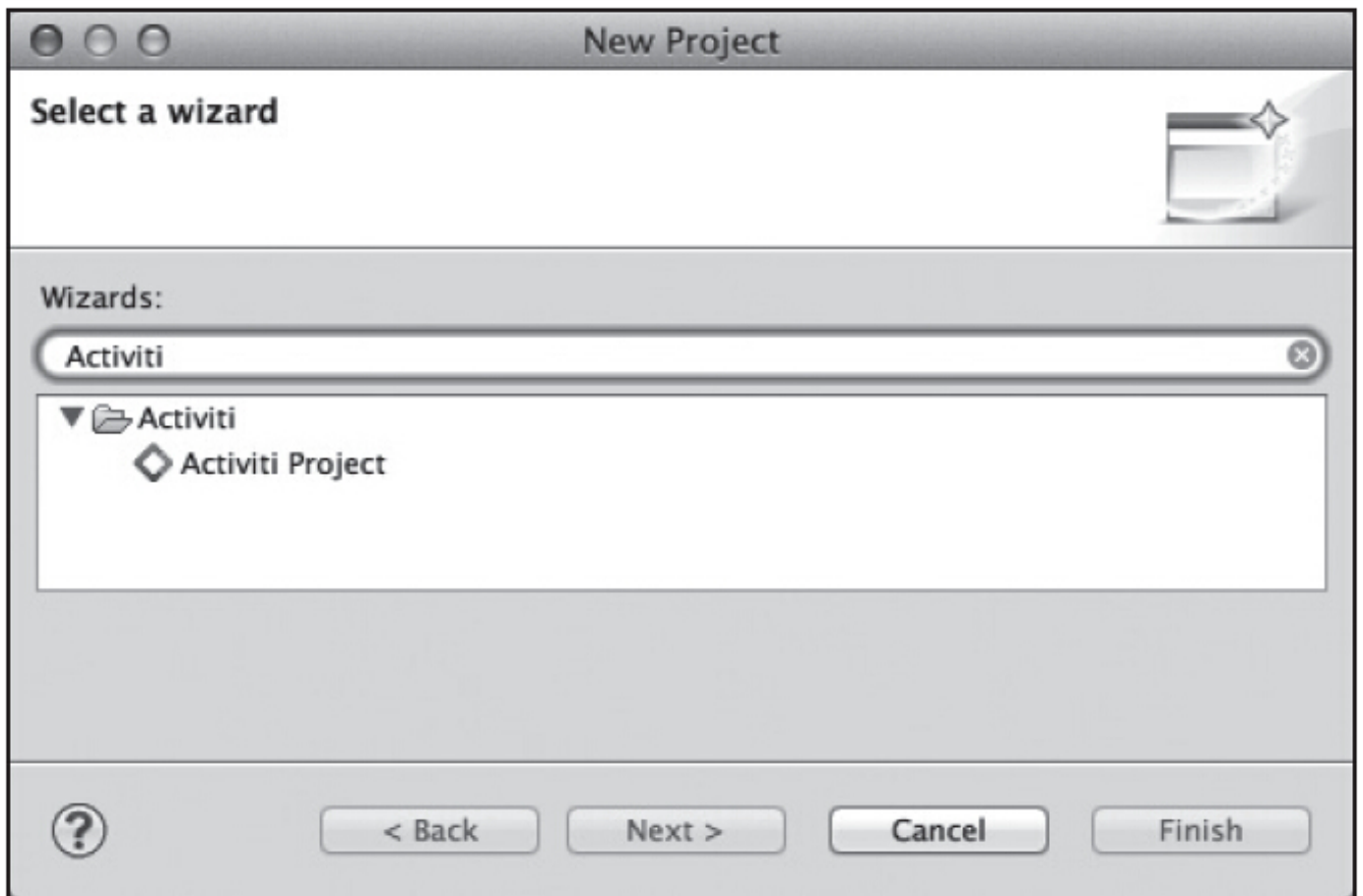


图3-25 新建Activiti Project向导

创建完项目后的目录结构如图3-26所示，其中展示的是一个名为“bpmn20-example”的项目，熟悉Maven的读者很快就看出这是Maven的标准目录结构。建议不熟悉Maven的读者参考Maven的文档了解相关知识。

新建的Activiti项目会自动在src/main/resources目录下创建diagrams包来存放新建的流程资源文件。在实际应用中往往习惯使用公司或组织域名倒序创建目录，如图3-26所示。

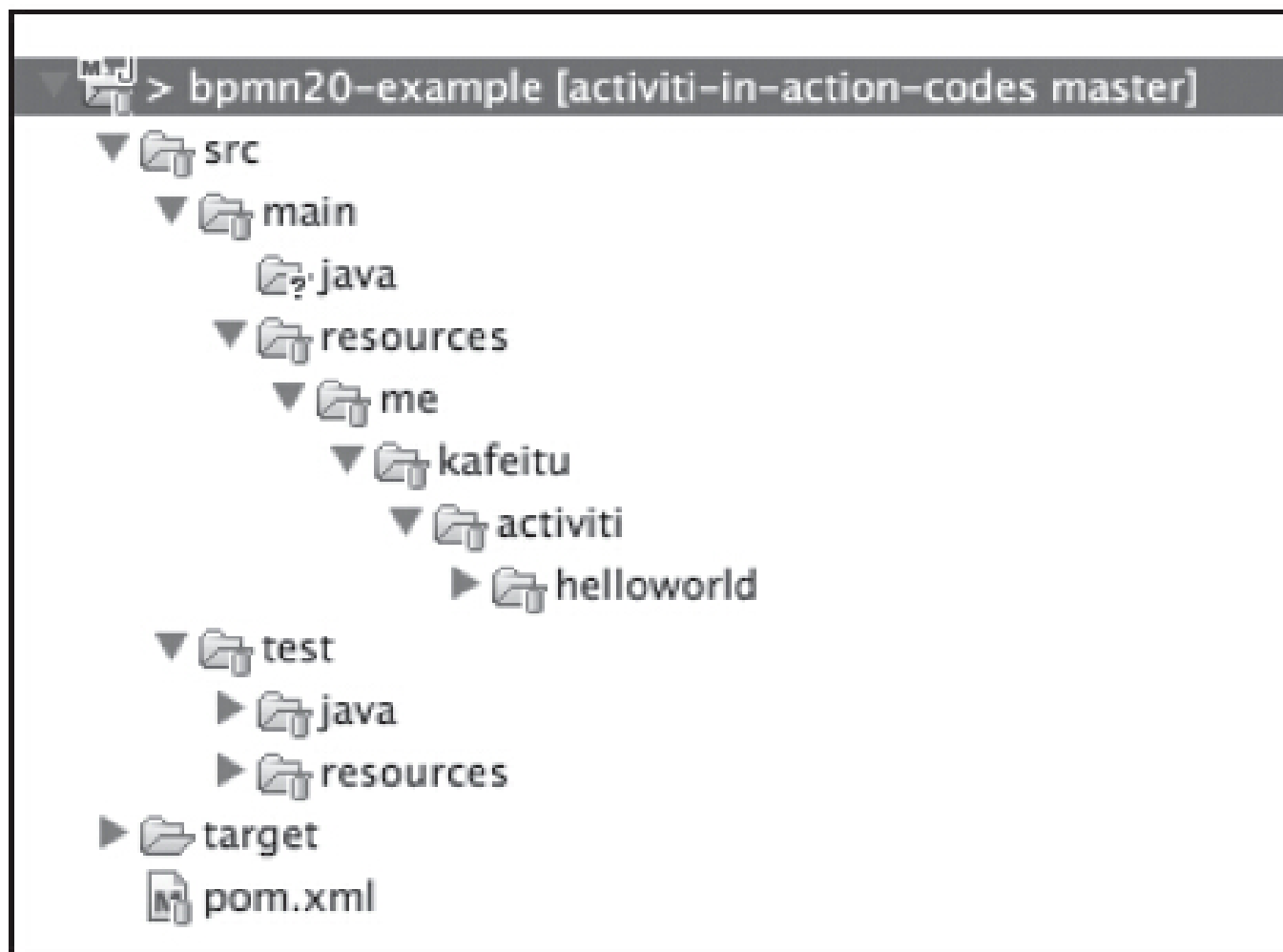


图3-26 Activiti项目的目录结构

2.新建请假流程

在“src/main/resources/me/kafeitu/activiti”目录上右击，选择“New”→“Other...”，打开如图片3-27所示的新建向导对话框，选择“Activiti Diagram”，单击“Next”按钮弹出如图3-28所示的对话框，其中默认的扩展名为**bpmn**，也是众多设计器默认的扩展名。

 **提示** 在Activiti Designer 5.8及之前的版本中，在新建流程定义时默认的扩展名为.activiti，而不是现在看到的.bpmn；当打开foo.activiti文件设计好流程并保存时会自动在同一目录下生成同名的foo.bpmn20.xml和foo.png。

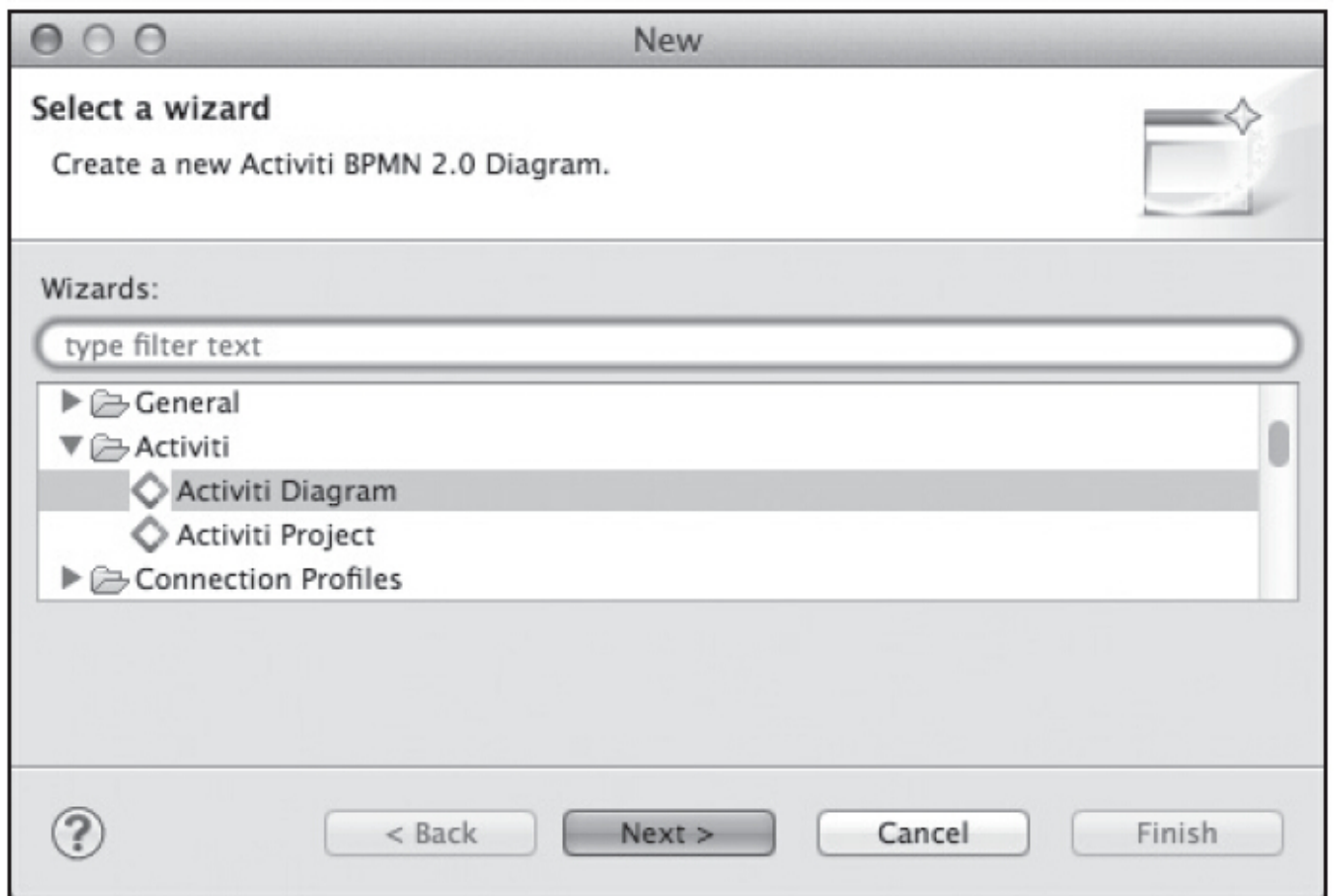


图3-27 新建Activiti Diagram向导——选择新建类型

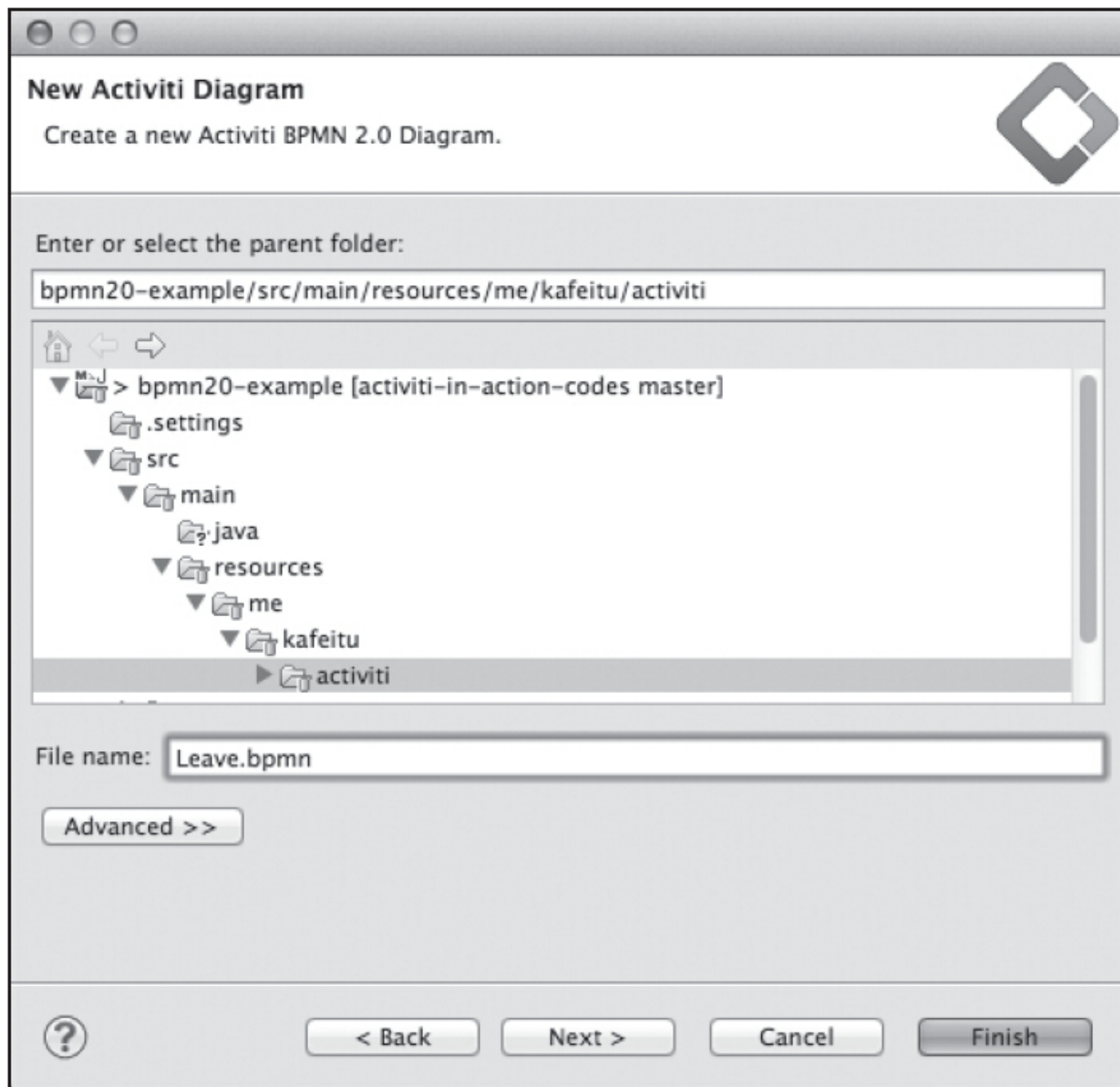


图3-28 新建Activiti Diagram向导——设置文件名

单击图3-28中的“Next”按钮之后弹出如图3-29所示的选择模板对话框。为了方便快速创建流程，Activiti Designer内置了一些流程定义模板供开发人员选择。在创建流程之后开发人员可以根据自己需求稍微调整即可。

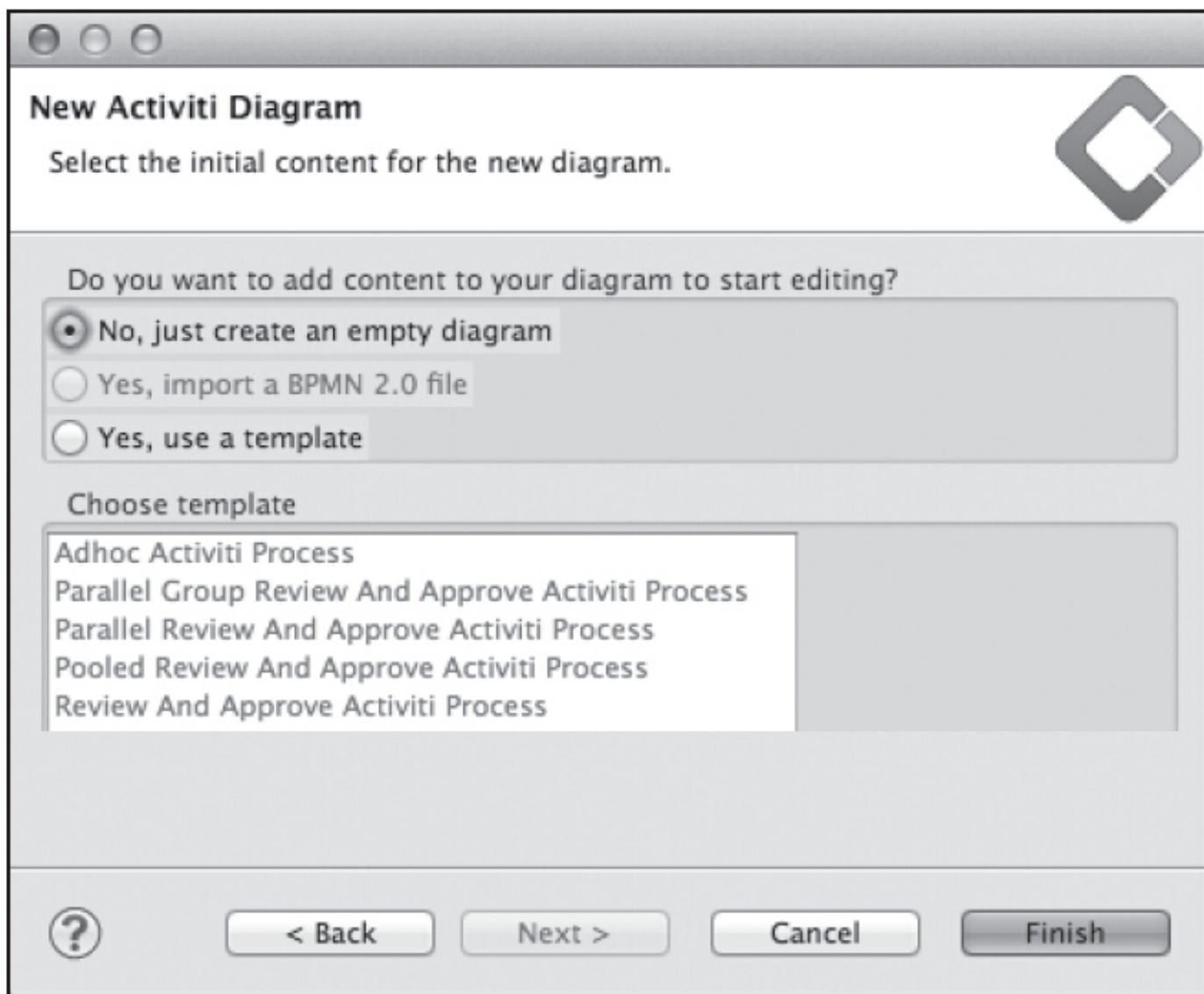


图3-29 新建Activiti Diagram向导——选择模板

提示 在安装Activiti Designer之后，在任何目录都可以通过“New”创建流程定义，前面关于创建Activiti Project的介绍仅仅是为了向读者说明创建过程。

3.设计请假流程

在新建流程之后要先设置流程属性，例如Id、Name等，设置请假流程的配置信息如图3-30所示。

下面介绍配置界面中的几个属性。

- **Id**: 流程的唯一标识，在2.4.2节中启动流程时指定的“leavehello”即此属性，建议使用纯英文标识。
- **Name**: 流程的名称，可以是任意字符。
- **Namespace**: 命名空间，一般使用公司名或组织域名+项目名称，可以更加细化到每个系统的模块，这样读者在实际项目中可以用此属性来归类流程，例如com/company/project/module，其中com/company是公司域名的倒序，project表示项目名称，module表示一个具体的模块；还可以使用另外一种描述方式，即通过<http://www.compnay.com/project/module>达到相同的目的。
- **Document**: 针对当前流程功能的简短文字描述。

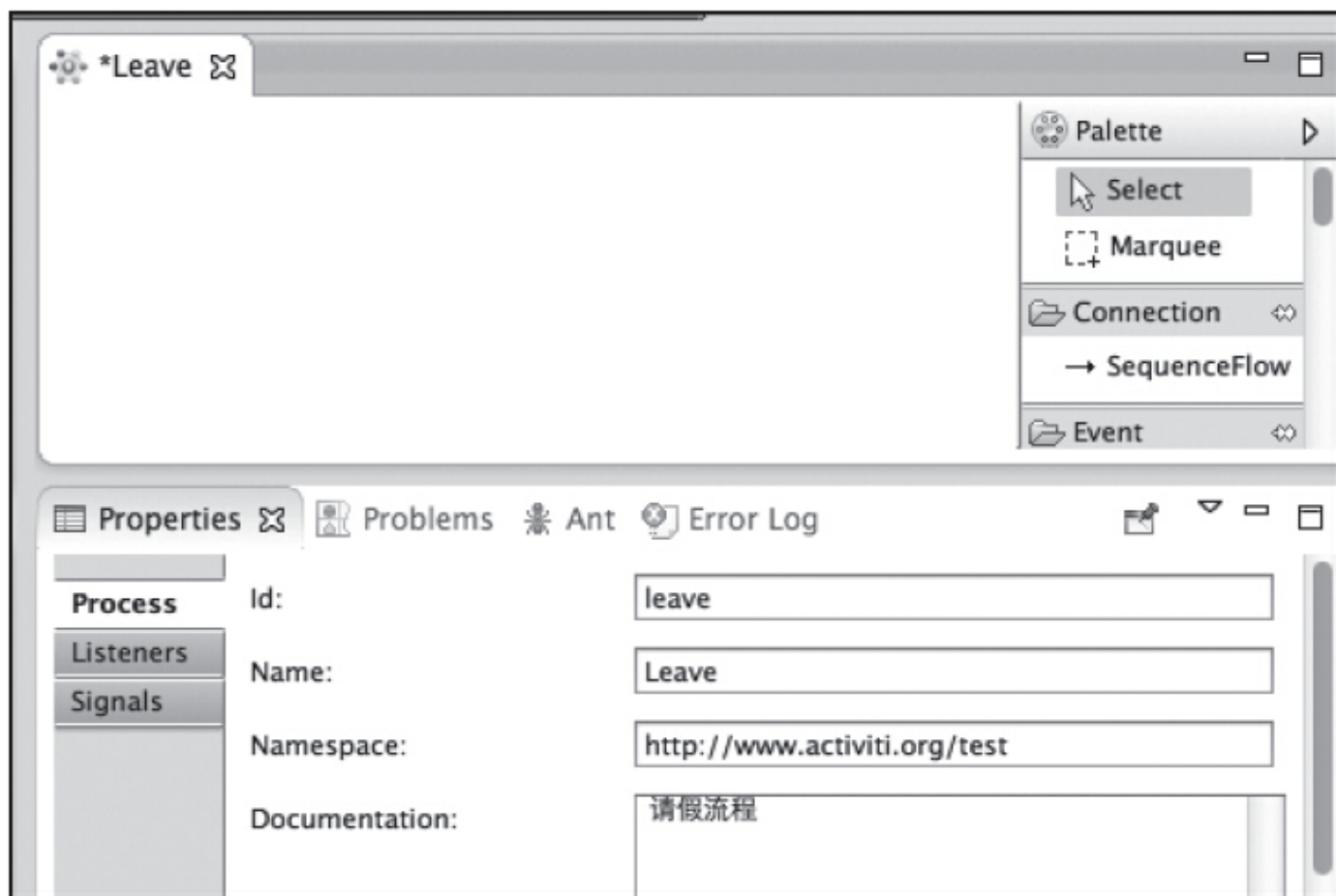


图3-30 设置流程属性

和Activiti Modeler设计流程的方式一样，从开始事件（Start Even）开始，在设计区域的右边栏中找到“Event”组的“StartEvent”，用鼠标将其拖动到左侧的空白区域，如图3-31所示。

图3-31中“Properties”下的Id、Name的含义和图3-30中的类似，只不过这里表示一个开始事件的属性。

在图3-31所示的工作区中，圆形图标即为开始事件，将鼠标移动到开始事件上之后和在使用Activiti Modeler时一样会浮动显示快捷方式菜单，例如可以通过单击左下的第二个图标创建一个任务（Task），如图3-32所示。

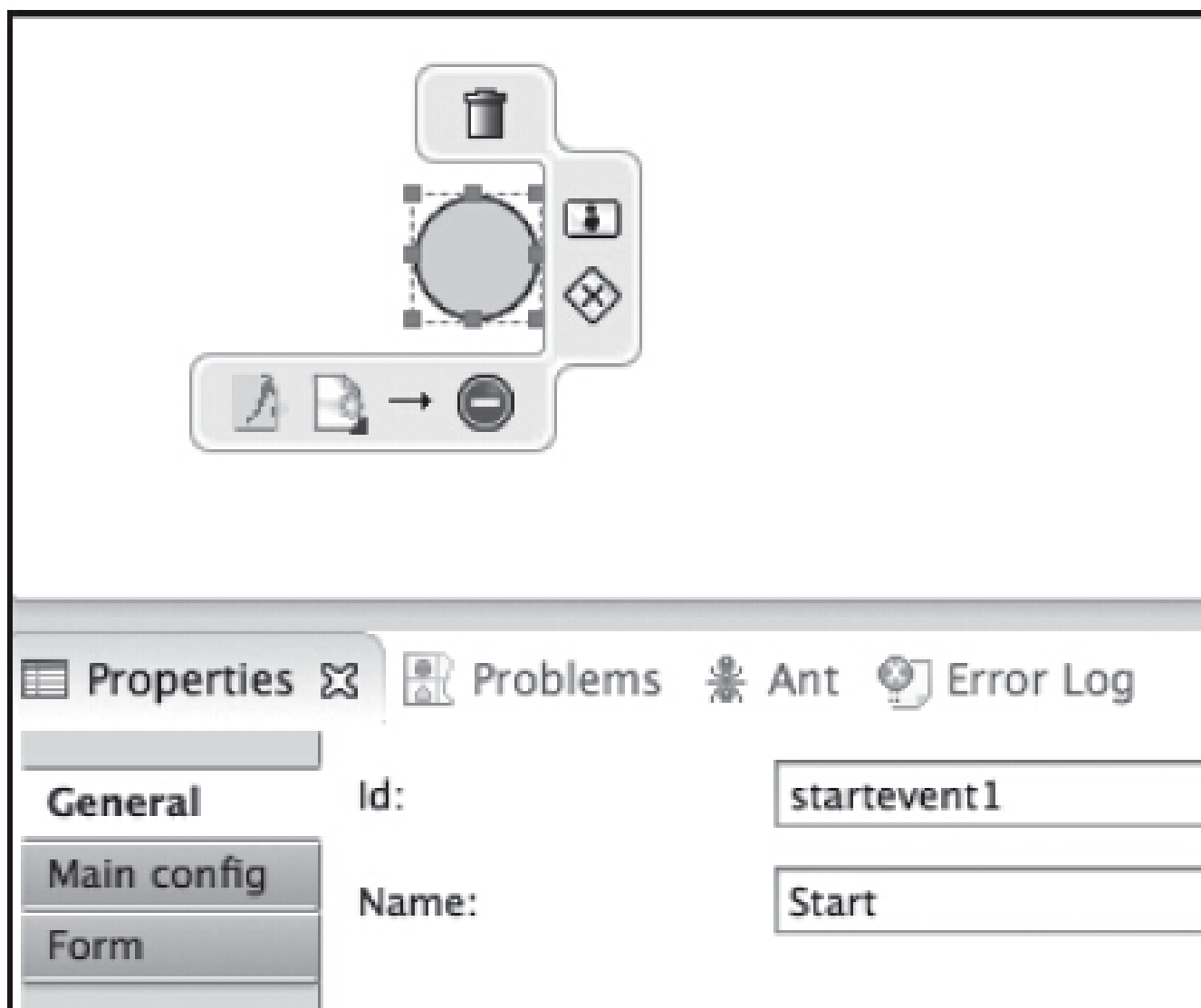


图3-31 开始事件

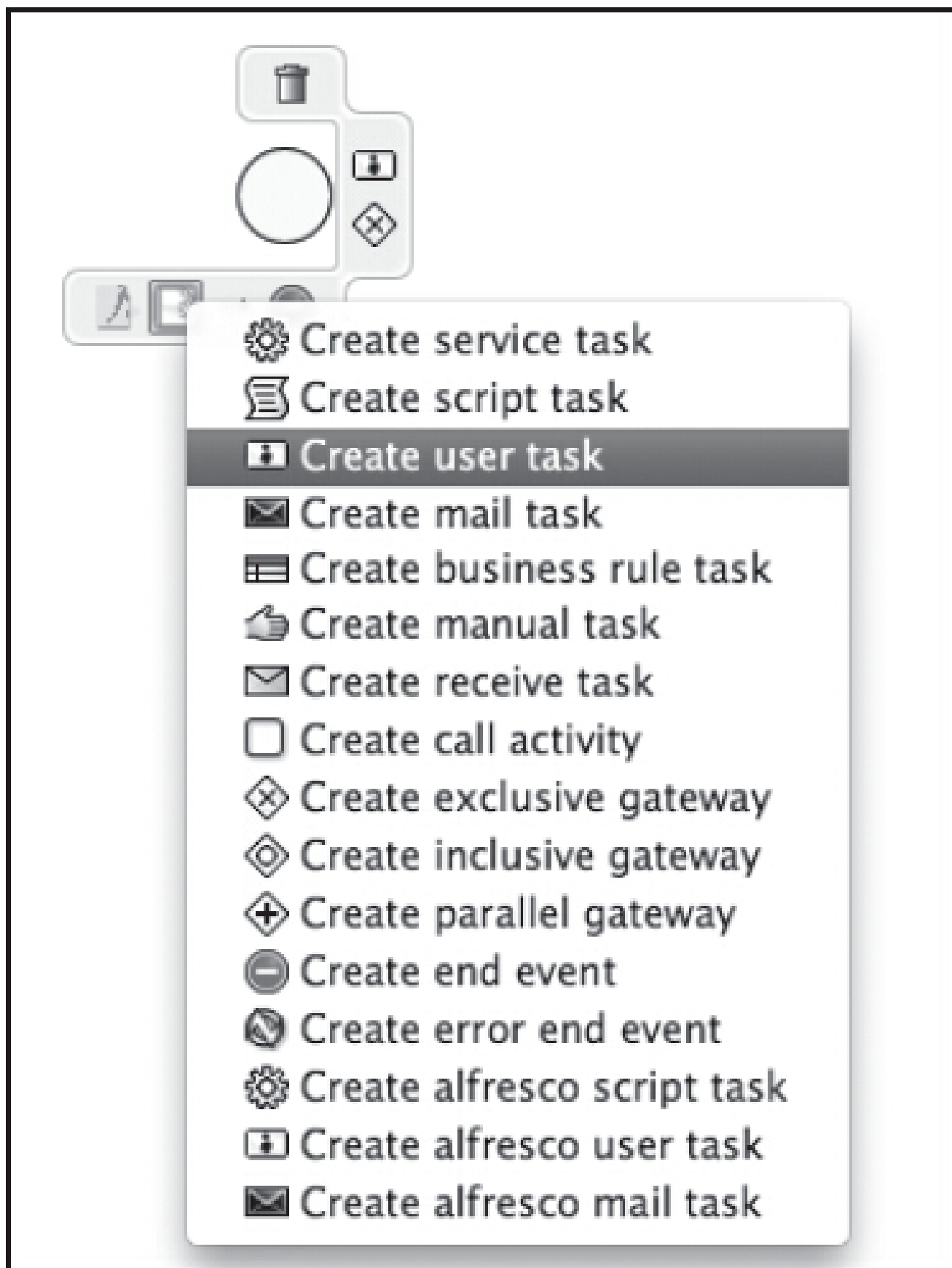


图3-32 创建开始事件

利用快捷菜单可以快速创建各种模型，当然也可以直接从右边栏的模型仓库中选择“Task”拖动到工作区。单击“Create user task”即可在开始事件的右边生成一个User Task并自动用顺序流（SequenceFlow）箭头连接，如图3-33所示。

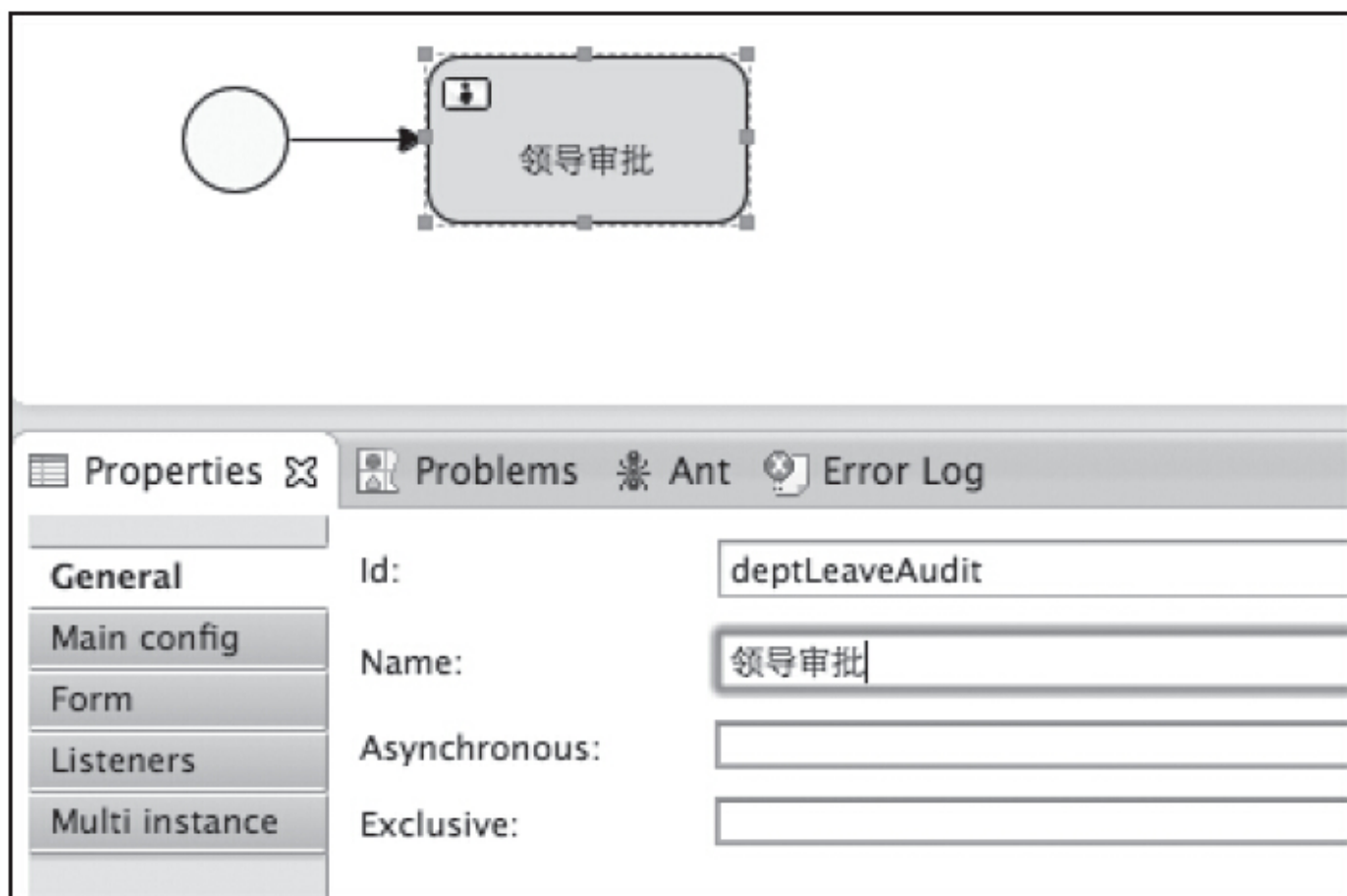


图3-33 通过快捷菜单创建User Task

在创建“领导审批”节点后单击刚刚新建的User Task就可以设置属性了，在图3-33中将Id、Name属性设置为“deptLeaderAudit”和“领导审批”，前者是Task在当前这个请假流程中的一个唯一标识。

设计“领导审批”任务在使用Activiti Modeler时已经结束了，在使用Activiti Designer时需要进一步设置任务的属性，例如任务的分配人。在图3-34中设置“领导审批”节点的分配人为“leader”。

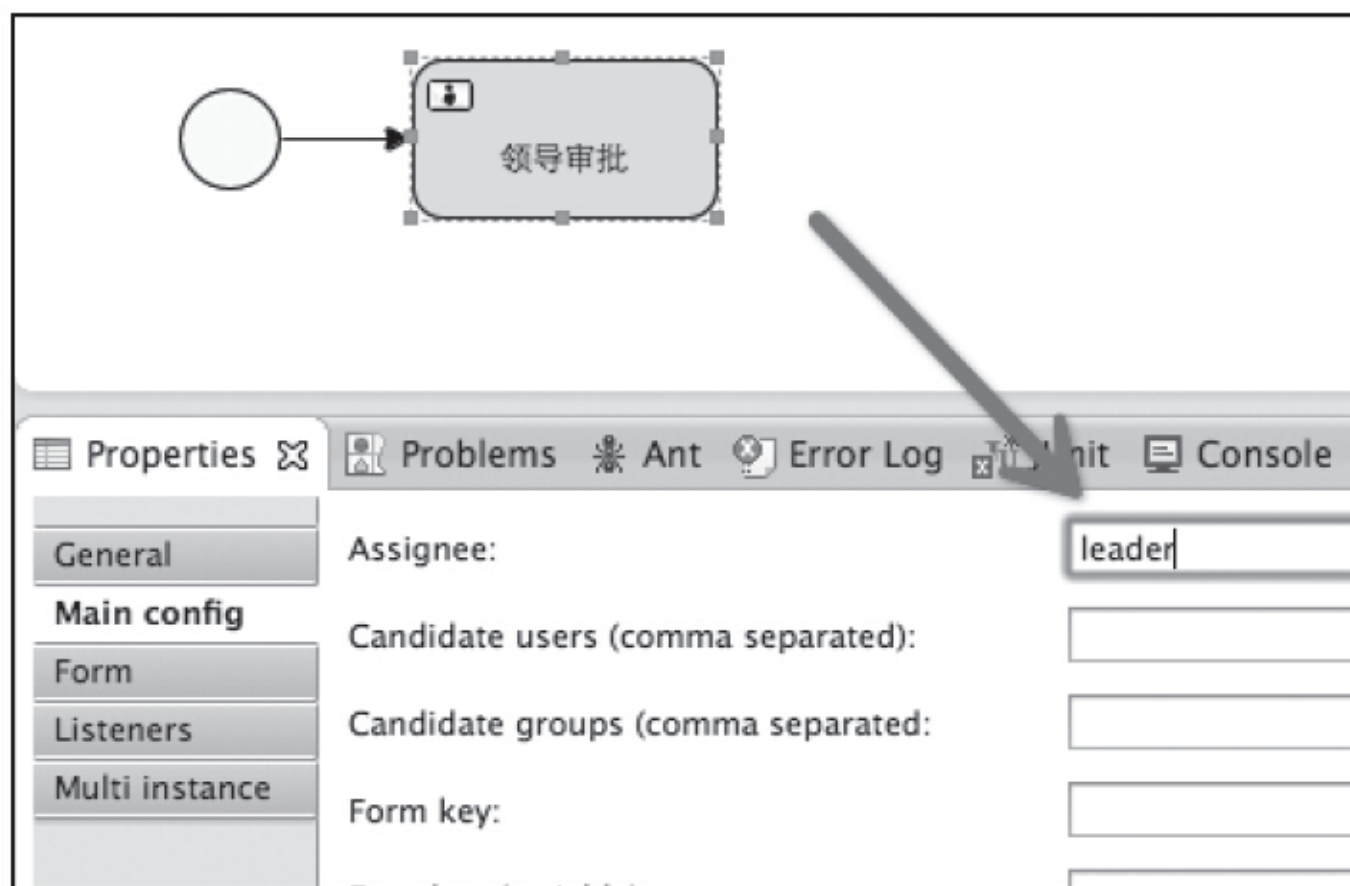


图3-34 设置User Task的任务分配人

读者可以自行依次完成剩余两个User Task的创建与设置，如图3-35所示。对于在设计过程中不容易理解的属性在以后的章节会陆续讲到，通过本章只要学会如何使用Activiti Designer设计简单流程即可。

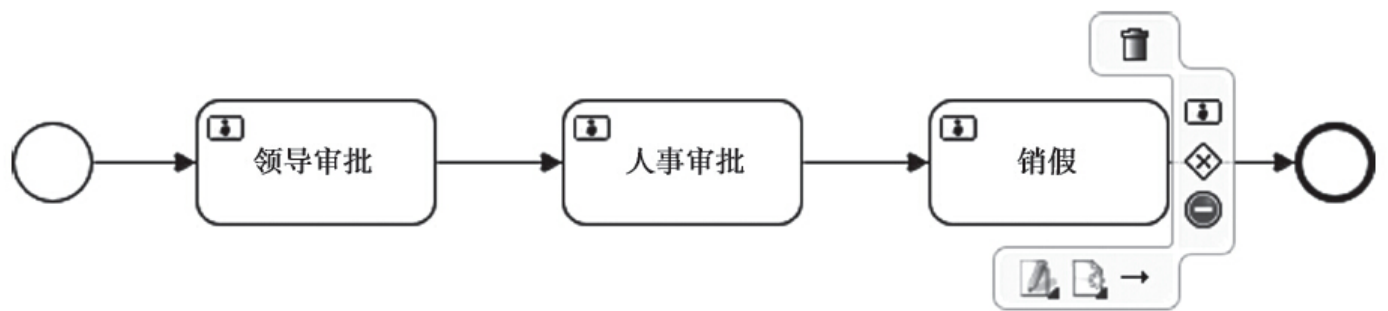


图3-35 使用Activiti Designer设计完成的请假流程

要在设计完流程后查看XML格式的文件内容，需要右击foo.bpmn文件选择“Open With” → “XML Editor”；要改用设计器打开要查看的文件，需要右击foo.bpmn文件选择“Open With” → “Activiti Digram Editor”。如图3-36所示，图中上半部分使用“Activiti Digram Editor”打开，下半部分使用“XML Editor”打开。

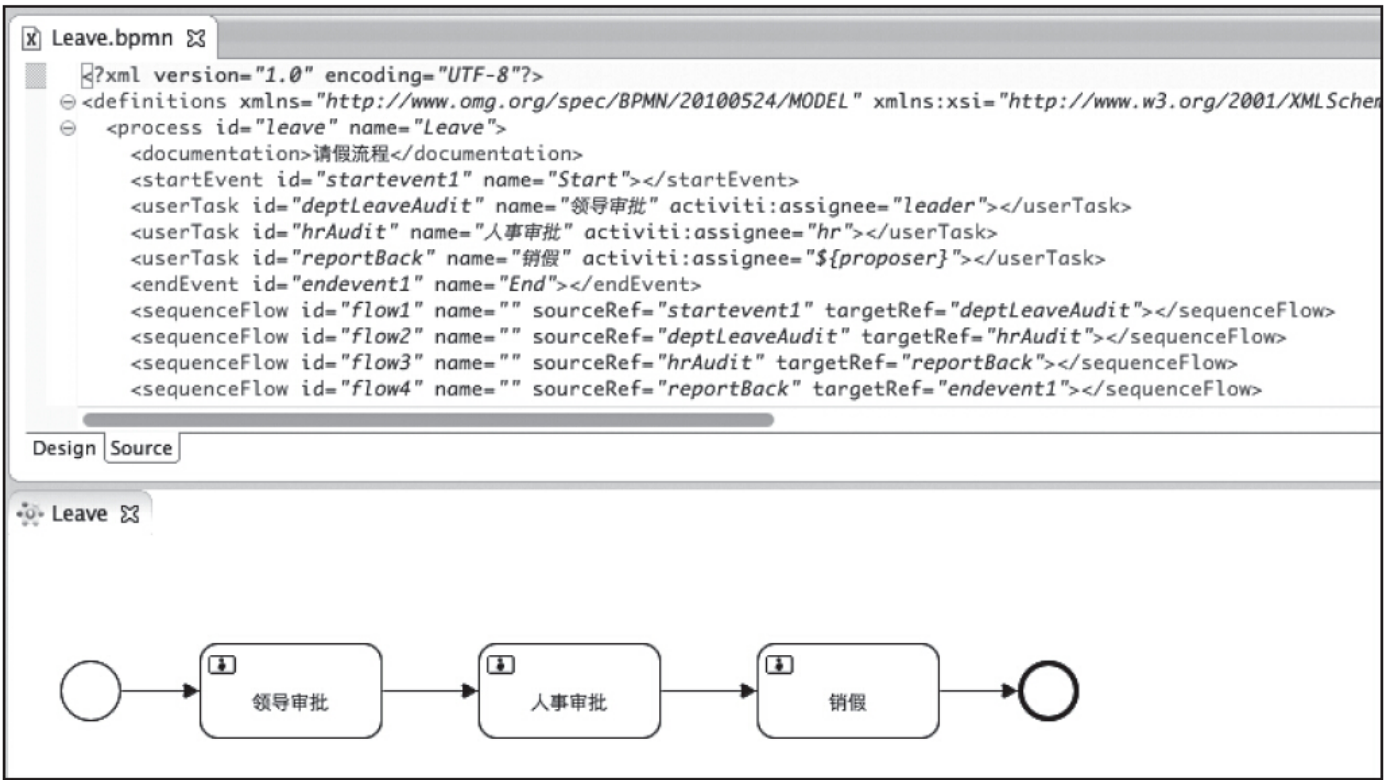


图3-36 使用不同方式打开Leave.bpmn文件

3.3.4 自动化

1.自动生成流程图片

流程图可以帮助用户快速了解整个流程的业务逻辑并根据不同的条件采用处理方式，在流程运行过程中可以以图片形式直观体现流程状态，例如标示当前在哪个节点可以用一个红色的边框表示。

Activiti Designer默认不会自动生成和流程相关的图片文件，需要用户自行更改插件的配置来启用自动生成图片功能。不同用户分别进行如下操作：

- Windows、Linux用户单击“Window”菜单选择“Preferences...”。
- Mac用户单击“Eclipse”菜单选择“Preferences...”。

打开如图3-37所示的配置对话框。依次选择对话框左边栏的“Activiti” → “Save”，勾选“Create process definition image when saving the diagram”前面的复选框。

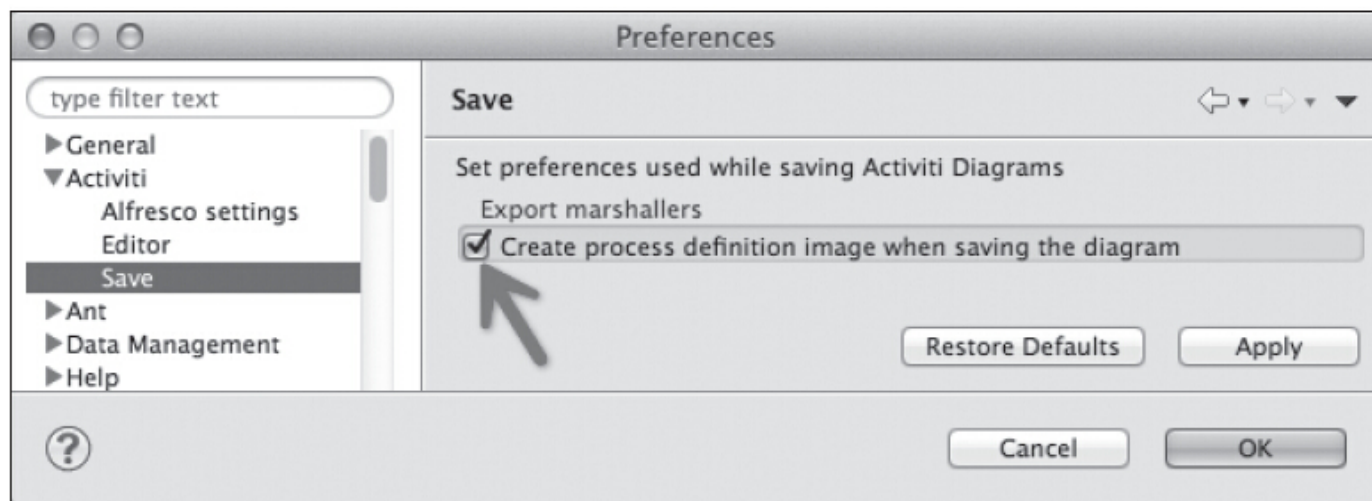


图3-37 设置自动生成图片功能

单击“OK”按钮，之后在新建流程或在原有流程中调整任意一个Task并进行保存即会在流程文件的同目录下生成同名的png图片文件，如图3-38所示。

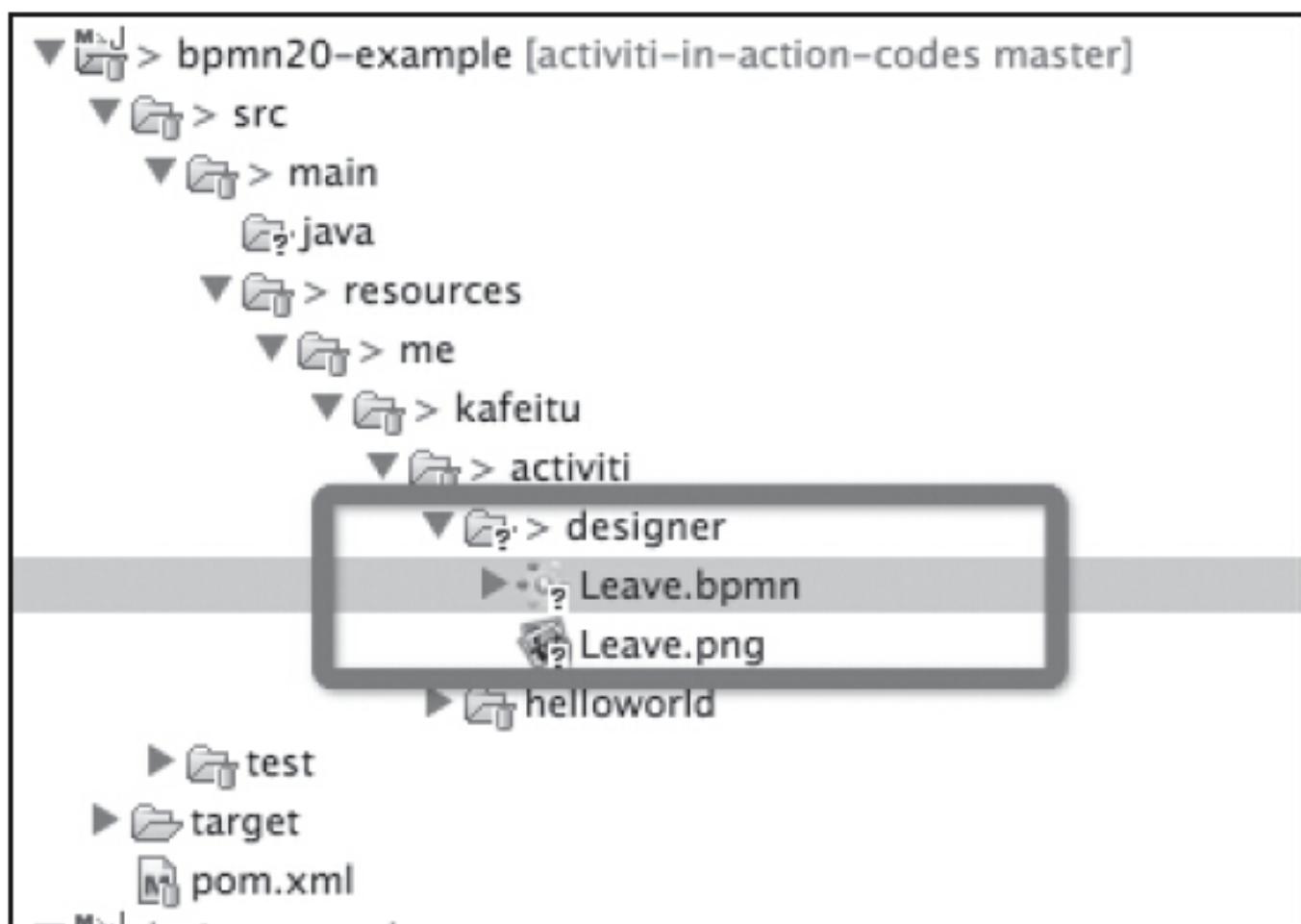


图3-38 自动生成的和流程同名的图片文件

2.自动生成测试代码

近几年敏捷开发在国内的企业和开源社区中被广泛使用，其中和开发者距离最近的当属极限编程（Extreme Programming，XP），而作为XP的核心原则之一的是测试驱动开发（Test Drive Development，TDD）。TDD能让代码更稳定、更健壮，使所有的新功能和重构都在TDD的基础上进行，把Bug扼杀在开发阶段而不是等到上线之后才发现。

Activiti引擎的开发也是使用TDD进行的，有兴趣的读者可以下载源码来阅读，其中的测试用例覆盖了大部分的引擎功能。同时Activiti Designer也为日常开发提供了便利的测试驱动的功能，使开发者可以很方便地基于流程定义文件生成对应的单元测试。

右击Leave.bpmn选择“Activiti”→“Generate unit test”，如图3-39所示。

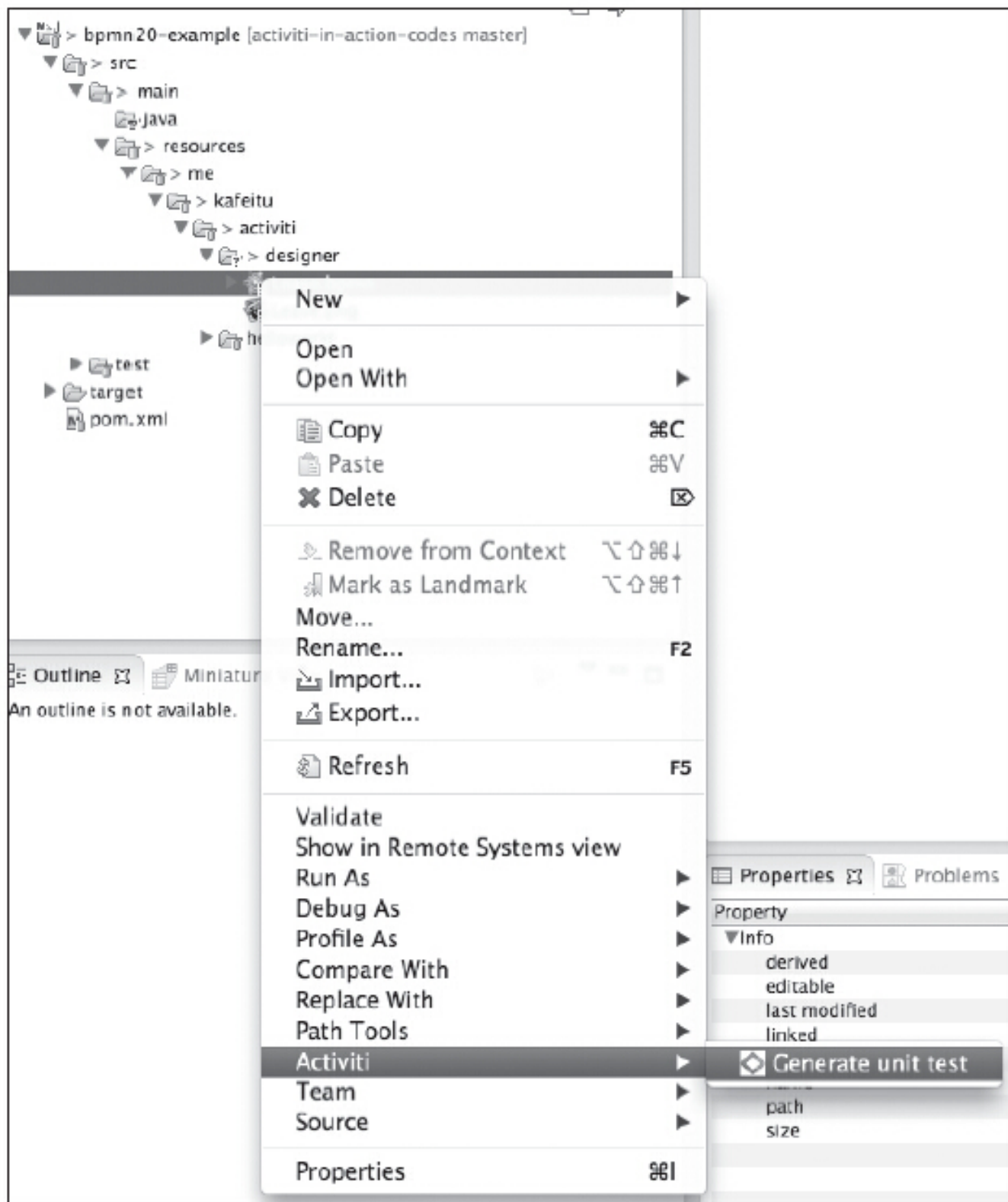


图3-39 自动生成单元测试的右键菜单

单击“Generate unit test”之后会在“test/java/org/activity/designer/test”包中生成“ProcessTestLeave.java”文件，如图3-40所示。

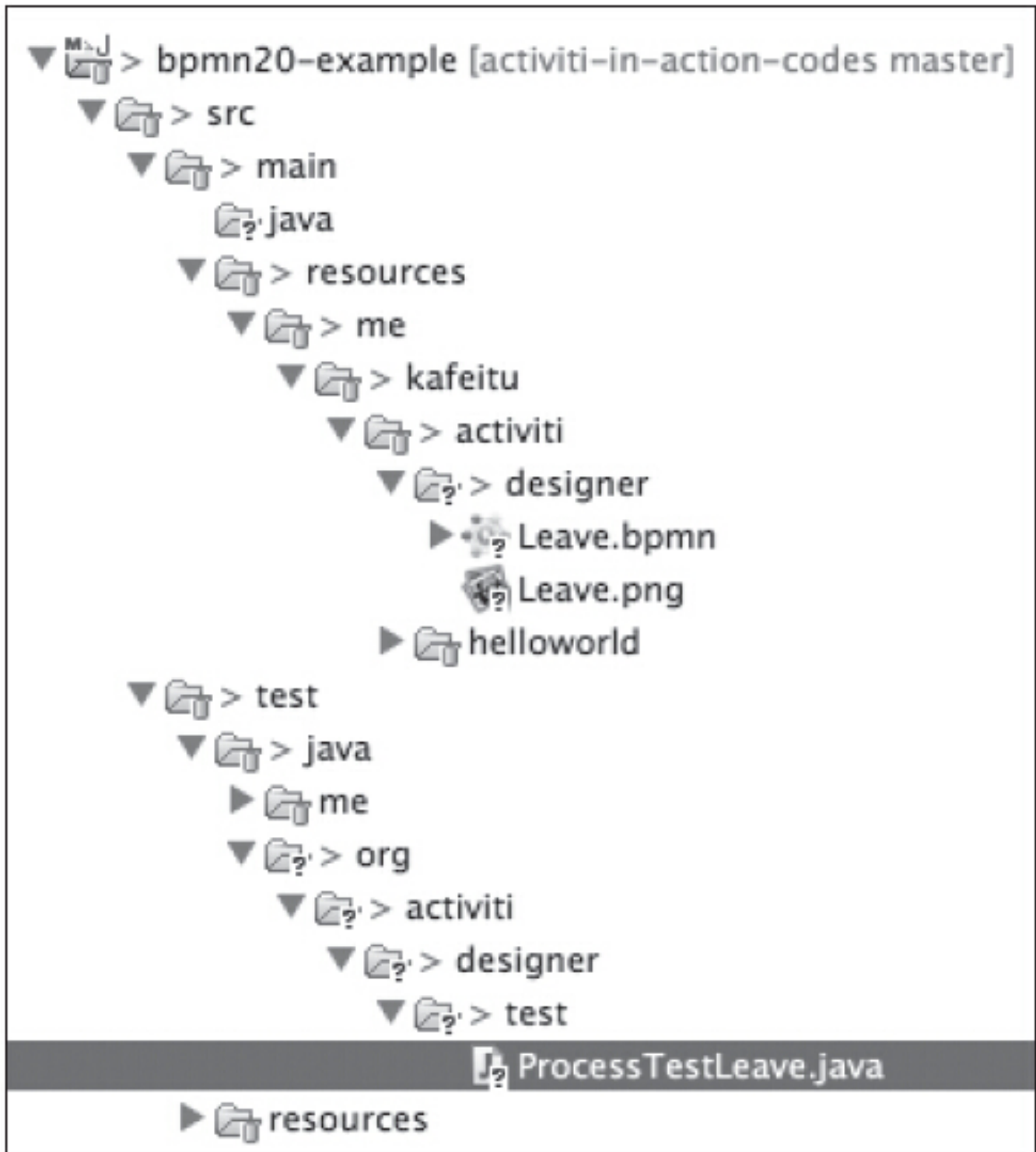


图3-40 自动创建的单元测试类

自动生成的测试代码见代码清单3-1。

代码清单3-1 Activiti Designer自动生成的测试代码

```
package org.activiti.designer.test;
// 省略了import部分
public class ProcessTestLeave {
    private String filename = "/Users/henryyan/work/books/activiti-in-action-codes/bpmn20-example/src/main/resources/me/kafeitu/activiti/designer/Leave.bpmn";
    @Rule
    public ActivitiRule activitiRule = new ActivitiRule(); #1
    @Test
    public void startProcess() throws Exception {
        RepositoryService repositoryService = activitiRule.getRepositoryService(); #2
        repositoryService.createDeployment().addInputStream( #3-S
            "leave.bpmn20.xml", new FileInputStream(filename)).deploy(); #3-E
        RuntimeService runtimeService = activitiRule
            .getRuntimeService(); #4-S
        Map<String, Object> variableMap = new HashMap<String, Object>();
        variableMap.put("name", "Activiti");
        ProcessInstance processInstance = runtimeService.startProcessInstanceByKey("leave", variableMap); #4-E
        assertNotNull(processInstance.getId()); #5
        System.out.println("#Id " + processInstance.getId() + " "
            + processInstance.getProcessDefinitionId());
    }
}
```

在代码清单3-1的#1处创建一个ActivitiRule，通过实例activitiRule可以获取第1章中提到的7个Service接口实例，如#2处获取了RepositoryService；接下来在#3-S至#3-E处以leave.bpmn20.xml作为

文件名部署流程；#4-S至#4-E处和代码清单2-3类似，启动了一个流程实例；最后在#5处验证流程实例processInstance的id属性不为空，进而证明流程启动成功。

有了单元测试类就可以用来验证流程的设计是否符合预期的结果，右键选择“Run As”→“JUnit Test”即可，如图3-41所示，运行结果如下：

```
Aug 2, 2012 11:51:37 PM org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
INFO: Loading XML bean definitions from class path resource [activiti.cfg.xml]
Aug 2, 2012 11:51:39 PM org.activiti.engine.impl.db.DbSqlSession executeSchemaResource
INFO: performing create on engine with resource org/activiti/db/create/activiti.h2.create.engine.sql
Aug 2, 2012 11:51:39 PM org.activiti.engine.impl.db.DbSqlSession executeSchemaResource
INFO: performing create on history with resource org/activiti/db/create/activiti.h2.create.history.sql
Aug 2, 2012 11:51:39 PM org.activiti.engine.impl.db.DbSqlSession executeSchemaResource
INFO: performing create on identity with resource org/activiti/db/create/activiti.h2.create.identity.sql
Aug 2, 2012 11:51:39 PM org.activiti.engine.impl.ProcessEngineImpl <init>
INFO: ProcessEngine default created
Aug 2, 2012 11:51:39 PM org.activiti.engine.impl.bpmn.deployer.BpmnDeployer deploy
INFO: Processing resource leave.bpmn20.xml
Aug 2, 2012 11:51:39 PM org.activiti.engine.impl.bpmn.parser.BpmnParser parseDefinitionsAttributes
INFO: XMLSchema currently not supported as typeLanguage
Aug 2, 2012 11:51:39 PM org.activiti.engine.impl.bpmn.parser.BpmnParser parseDefinitionsAttributes
INFO: XPath currently not supported as expressionLanguage
id 5 leave:1:4
```

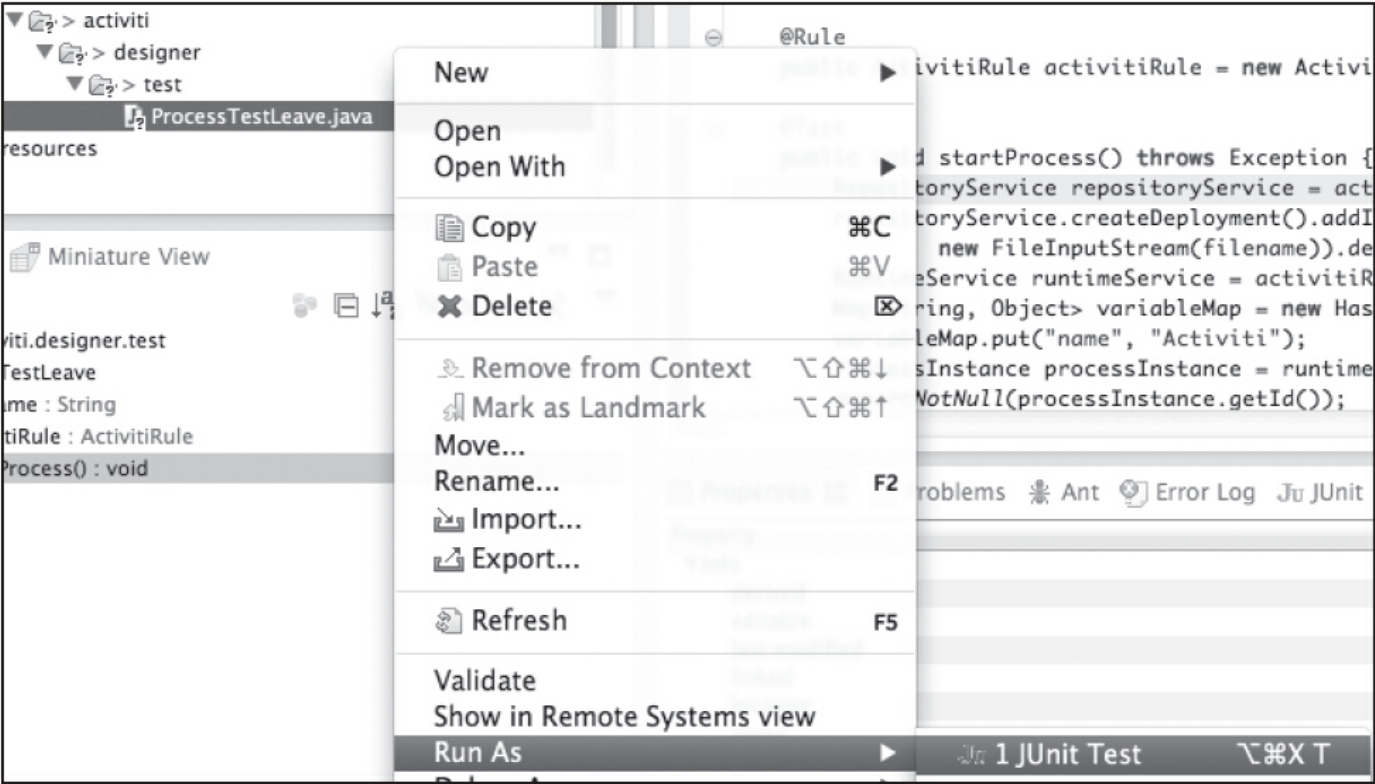


图3-41 运行单元测试

读者可以通过自动生成测试代码功能来学习Activiti，当想了解Activiti如何处理一个流程组件时可以先设计一个流程定义，然后生成单元测试代码，利用JUnit的功能验证引擎运行的结果是否和预期一致。

3.3.5 升级历史遗留的流程设计

第2章的Hello World是在部署XML格式的文件之后启动流程的。使用Activiti Modeler设计的流程定义会在工作区生成两个文件：foo.bpmn20.xml和foo.signavio.xml，这两个文件的作用前面已经讲到了，这里不再重复。那么在Activiti Designer中如何生成foo.bpmn20.xml文件呢？

答案很简单，用于设计流程的foo.bpmn文件就是我们想要的foo.bpmn20.xml，为什么这么说呢？这既是业界的约定，也有一定的历史原因。

bpmn是BPMN XML文件的默认扩展名。很多设计器都使用bpmn作为流程定义文件的扩展名。

Activiti Designer 5.8及之前的版本是将.activiti作为设计文件存在，在保存foo.activiti之后会自动生成foo.bpmn20.xml和foo.png两个文件。之后在Activiti Designer 5.9版本发布时放弃了这一做法改用标准的 bpmn 作为扩展名，同时不会再生成foo.bpmn20.xml文件，因为新的.bpmn文件内容和Activiti Designer 5.8版本之前自动生成的foo.bpmn20.xml文件内容一样。

升级Activiti Designer 5.8及之前版本的设计文件至兼容新版本的步骤如下：

- 1) 删除.activiti文件，不再需要它了。
- 2) 重命名foo.bpmn20.xml为foo.bpmn，刚刚已经提到到，文件内容相同，仅仅是扩展名不同而已。

3.3.6 导入Activiti Modeler设计

在3.1节中了解了如何使用Activiti Modeler设计流程，并且也看到了生成的leave.bpmn20.xml文件，现在假设业务流程设计已经完成，要将其转移给开发人员进行完善以满足引擎运行的要求。

把Activiti Modeler工作区中的leaven.bpmn20.xml复制到项目中，将其重命名为leave.bpmn，双击leave.bpmn就可以以设计器方式将其打开以便更改流程配置，如图3-42所示。

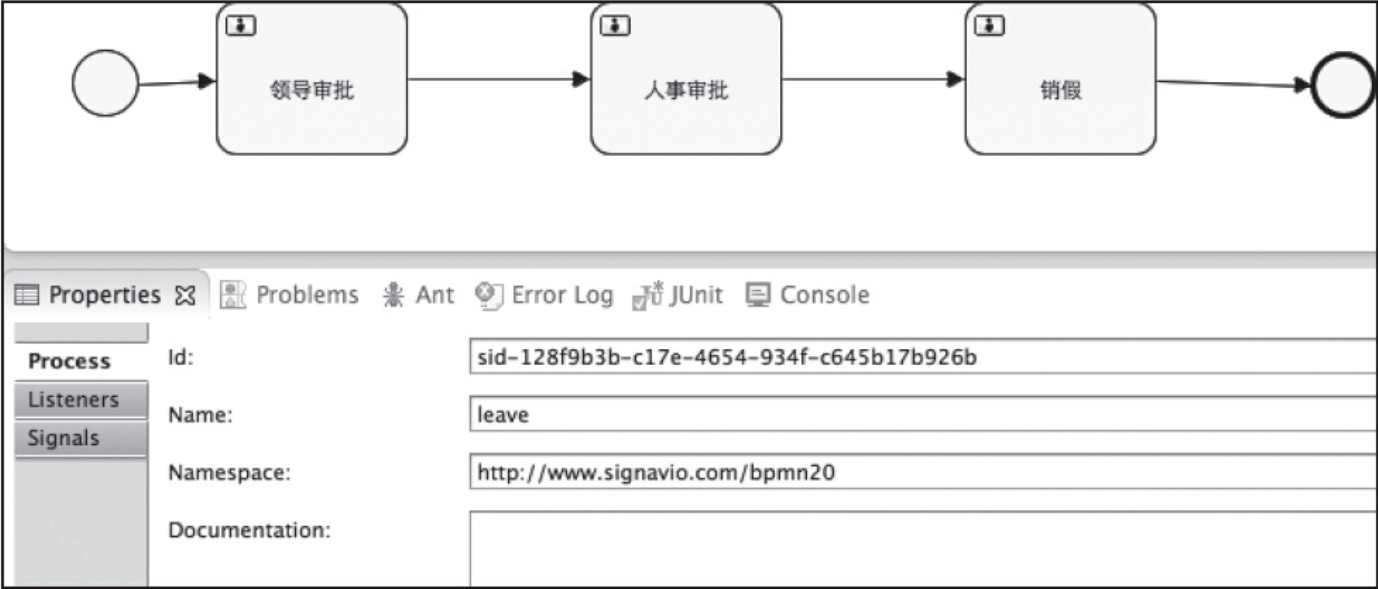


图3-42 将使用Activiti Modeler设计的流程定义导入至Eclipse工程中

至此可以执行像3.3.3节“设计流程”那样的过程来完善流程定义信息，例如可以将一长串的字符表示的流程的ID属性更改为leave，完成之后同样可以生成单元测试功能来验证更改是否正确。

3.3.7 泳池与泳道

图3-43是一个常见的标准游泳池，一个泳池（Pool）中分为了多个泳道（Lane），比赛时每一个参赛选手只能在自己的泳道中前行。

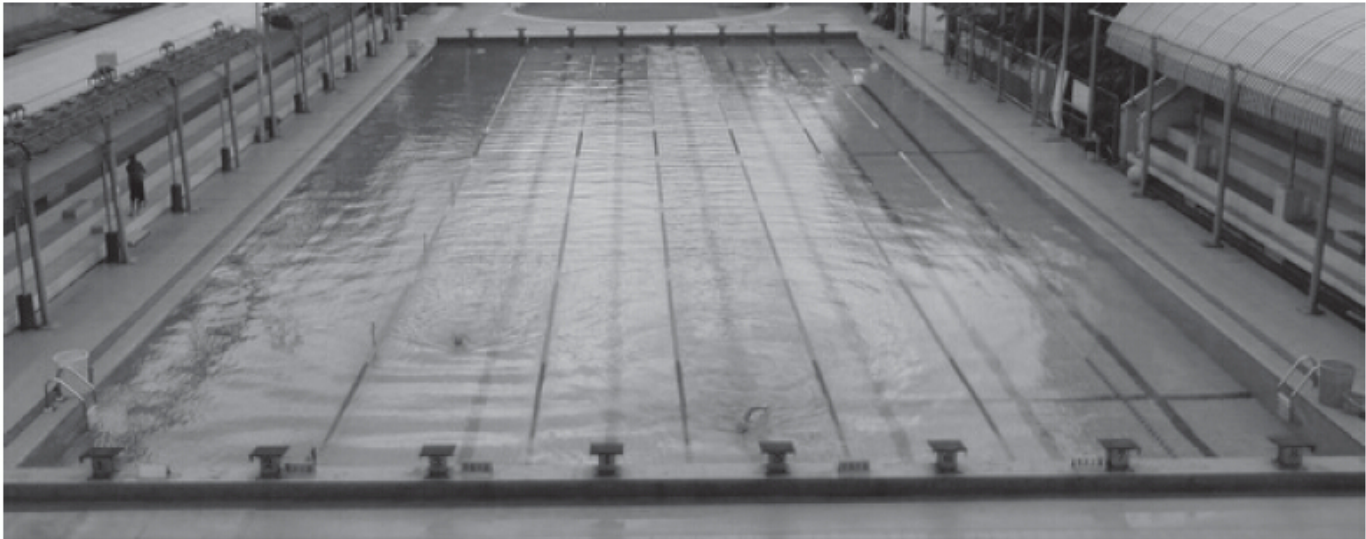


图3-43 游泳池

相信有一部分读者使用过Visio或类似的工具来设计流程，为了沟通方便和清晰展示不同角色、岗位人员的职能，就会使用泳池与泳道来划分。
回想一下请假流程中的设计，涉及的人员角色分别有：员工、部门领导、人事，在BPMN 2.0规范中请假流程还可以这么来设计，如图3-44所示。

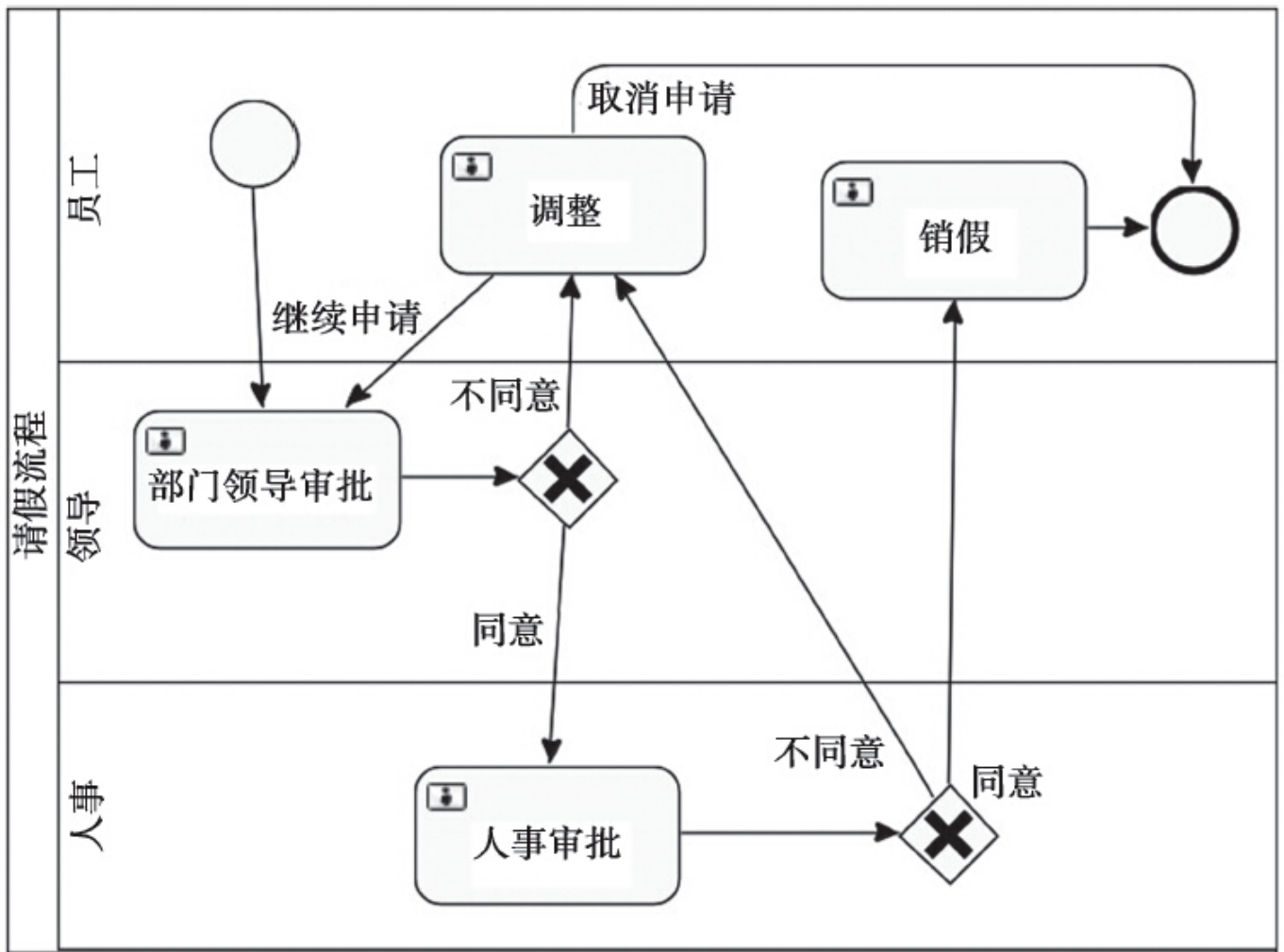


图3-44 请假流程的泳道模式

读者可以把前面章节接触到的请假流程与图3-44进行对比，显然图3-44中的泳道模式会更加直观、清晰地划分出不同角色、职能人员所需要负责的任务。

图3-44中包含了一个泳池（名称为：请假流程）以及3个泳道（员工、领导、人事）。

3.3.8 在Activiti Designer中使用泳道与泳池

新建一个流程，然后在打开的设计窗口右侧选择“Container”栏中的“Pool”并拖动到空白区域，如图3-45所示。

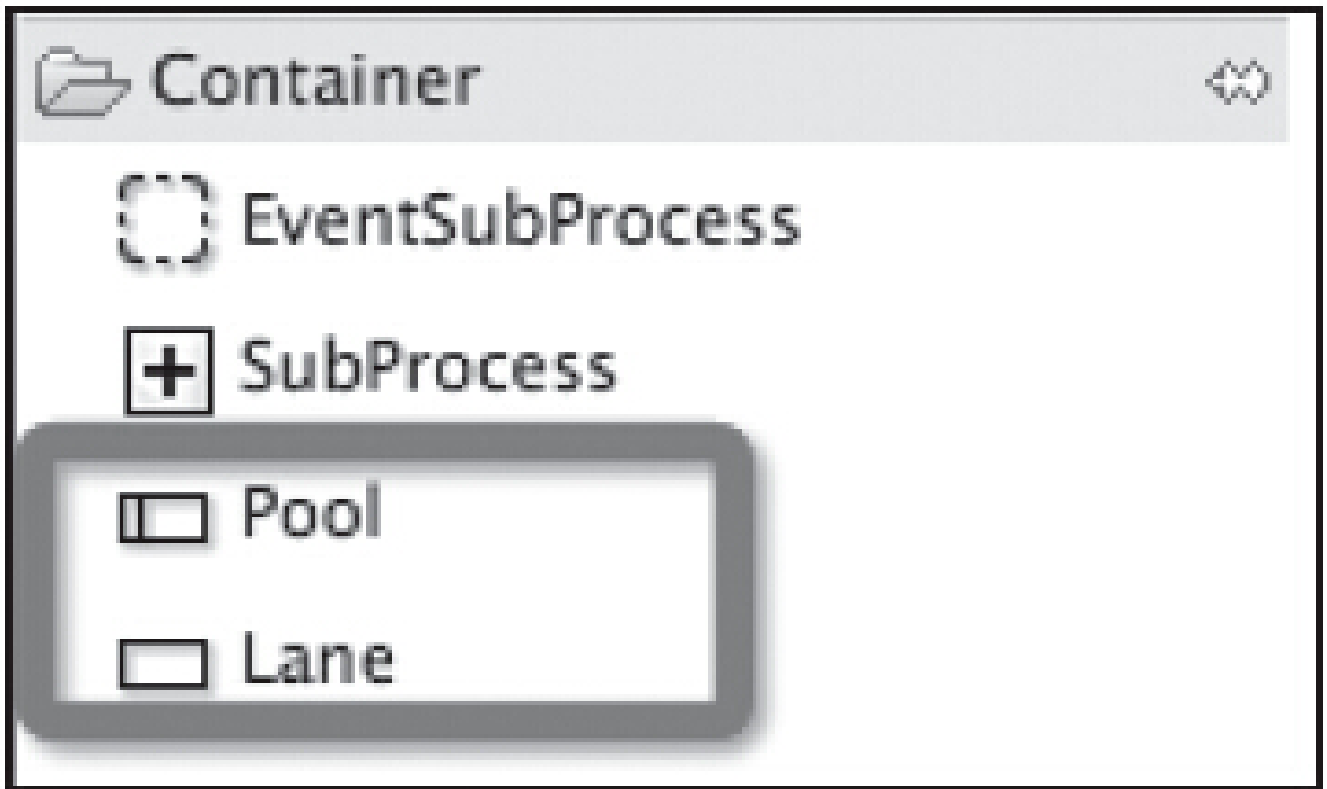


图3-45 从工具栏中选择泳池与泳道

拖动完成后即创建了如图3-46所示的泳池，此时可以单击泳池左侧（Pool文字处）选中泳池以编辑其属性，如图3-47所示，把泳池的名称改为“请假流程”。



图3-46 只包含一个泳道的泳池

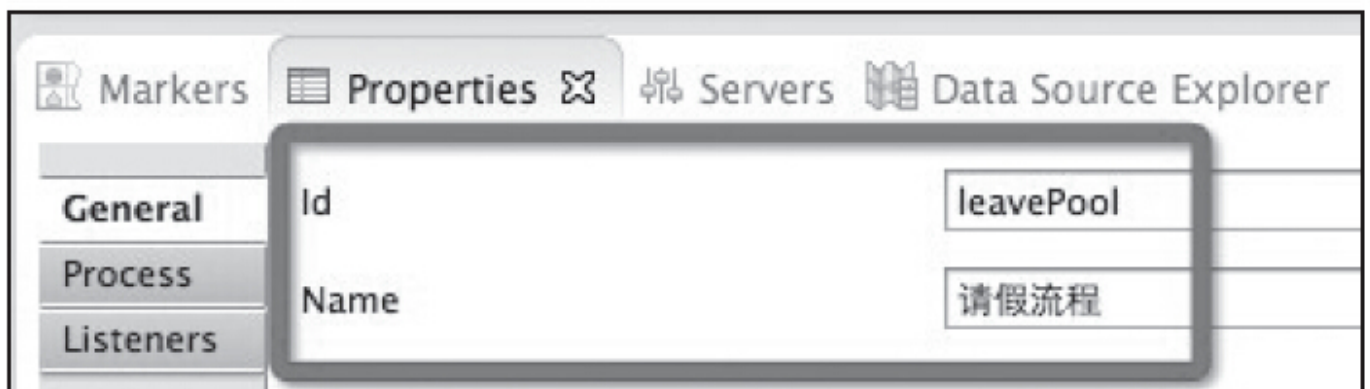


图3-47 编辑泳池的属性

单击图3-46中的泳道同样可以编辑属性，把第一个泳道命名为“员工”用来存放和请假申请人有关的任务，如图3-48所示。

接下来就可以像前面章节一样来设计流程，在“员工”泳道中拖动一个启动事件，如图3-49所示。

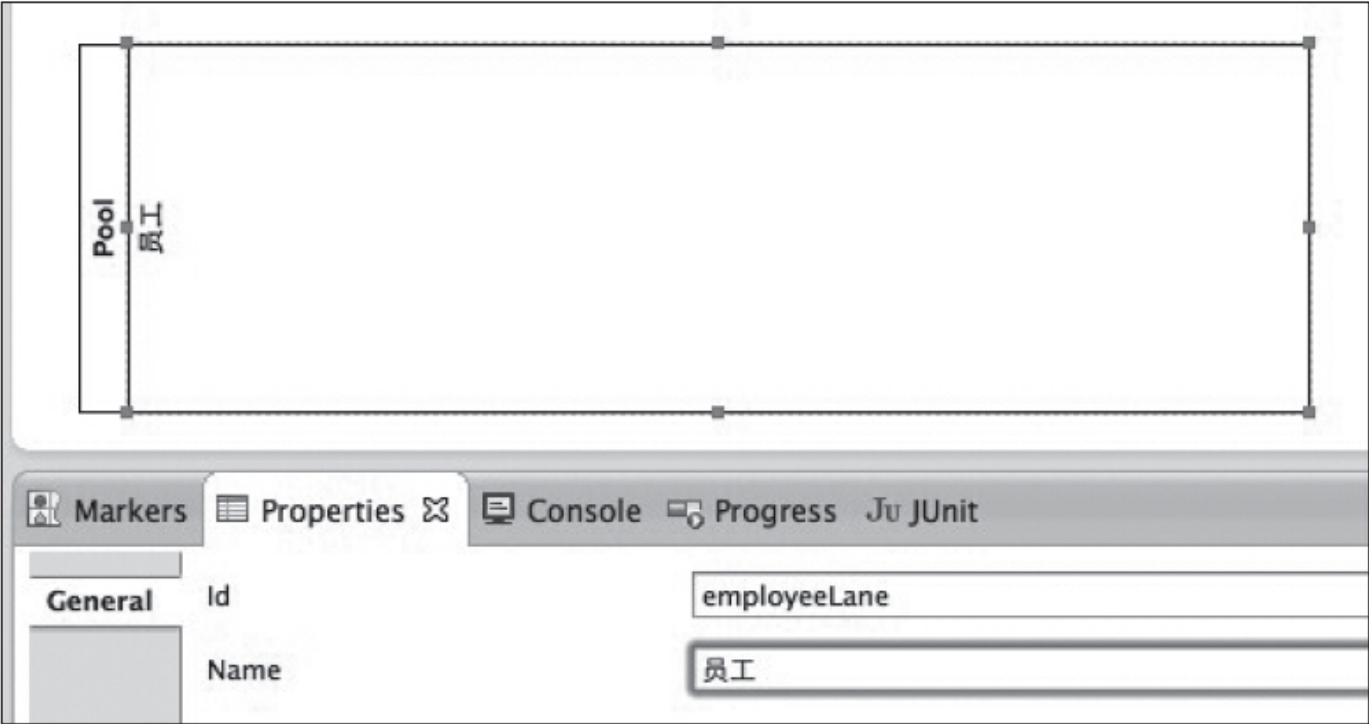


图3-48 选择并编辑泳道属性



图3-49 在员工泳道中拖入一个空启动事件

接下来需要再创建一个针对领导的泳道，在右侧工具栏选择“Container”中的“Lane”拖动到泳池中，如图3-50所示。

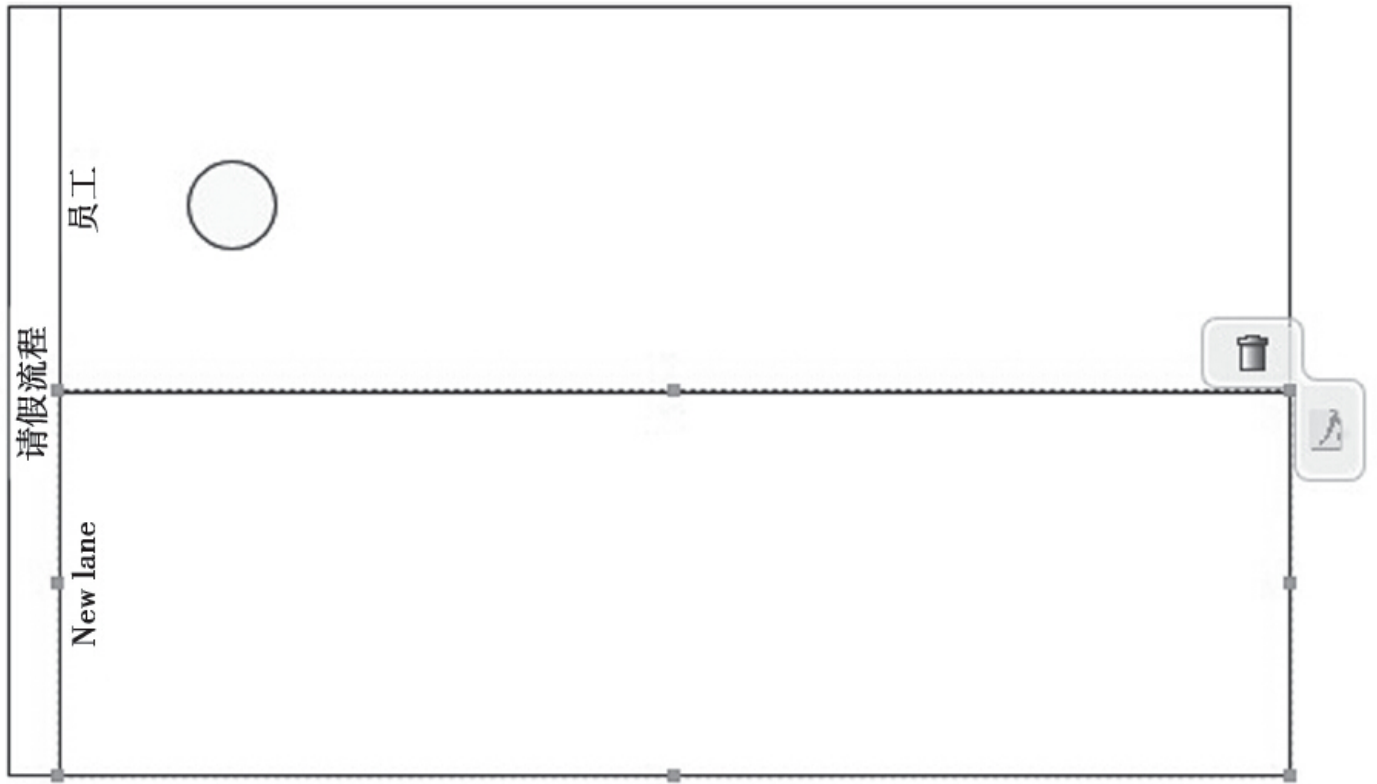


图3-50 创建了一个新的泳道

以此类推读者就可以设计出如图3-44所示的流程图。

物理的泳池只能划分固定数量的泳道，在设计流程图时对泳道的数量没有限制，可以由任意多个泳道组成一个大的泳池，所以泳道与泳池常在一些相对复杂的流程中使用。

3.4 本章小结

“工欲善其事，必先利其器”。本章主要介绍了Activiti引擎提供的两个流程设计器，一个是由Signavio授权的基于Web的Activiti Modeler，另外一个则是Activiti Designer。两个设计器侧重面不同，Activiti Modeler偏向于业务层面，而Activiti Designer则更侧重于开发层。

针对Activiti Modeler本章讲解了如何进行打包、部署、运行等，并且简单介绍了如何使用Activiti Modeler设计一个简单的请假流程。针对Activiti Designer本章详细讲解了安装插件、设计流程、自动生成图片和单元测试代码等内容，又介绍了针对历史原因致使流程定义文件扩展名不同而进行的升级设计，最后介绍了从业务层向开发层导入流程定义文件的方法。

有了对流程引擎的初步认识及使用工具设计流程的基础，对于下一章将详细介绍的目前Activiti支持的BPMN 2.0规范，读者可以借助Activiti Designer更好地学习。

第4章 Activiti与BPMN 2.0规范

理论是实战的基石，因此详细了解Activiti支持的BPMN 2.0规范显得尤为重要，尤其是对前面章节示例中的模型定义不太了解的读者更要了解BPMN 2.0规范。

BPMN 2.0规范包含很多种模型。目前Activiti可以支持在大多数情况下常用的模型，并且在实现规范的基础上进行了功能和使用性扩展。在讲解的过程中首先会介绍BPMN 2.0规范中的定义方式，如果Activiti添加了扩展属性、配置，也会介绍其含义及配置方法。

本章每一节将按照概念、图形、XML描述、实例讲解及Activiti的扩展属性的顺序进行讲解，理论结合实例使读者快速对BPMN 2.0规范有一个全面的认识；有BPMN 2.0规范基础的读者可以了解Activiti针对BPMN 2.0规范的封装及扩展，借助Activiti的扩展功能可以更方便地实现和BPMN 2.0规范中相同的功能，并且通过简单配置即可实现邮件任务、Java Service任务等不在BPMN 2.0规范之内的功能。

本章内容根据BPMN 2.0规范的分类划分为以下几部分：

- 启动与结束事件（Event）
- 顺序流（Sequence Flow）
- 任务（Task）
- 网关（Gateway）
- 子流程（Subprocess）
- 边界事件（Boundary Event）
- 中间事件（Intermediate Event）
- 监听器（Listener）

4.1 启动事件与结束事件

启动事件与结束事件作为BPMN 2.0规范中很重要的一部分，分别负责一个流程的开始和结束。一个完整的流程可以分为两大类：启动事件、结束事件。每一个大类又可以根据功能不同分为多个小类。

所有的开始事件均以如图4-1所示的细线圆形为基础。本节讲解的其他事件都是在其基础上“嵌入”对应类型的图形来表示一种启动事件的。

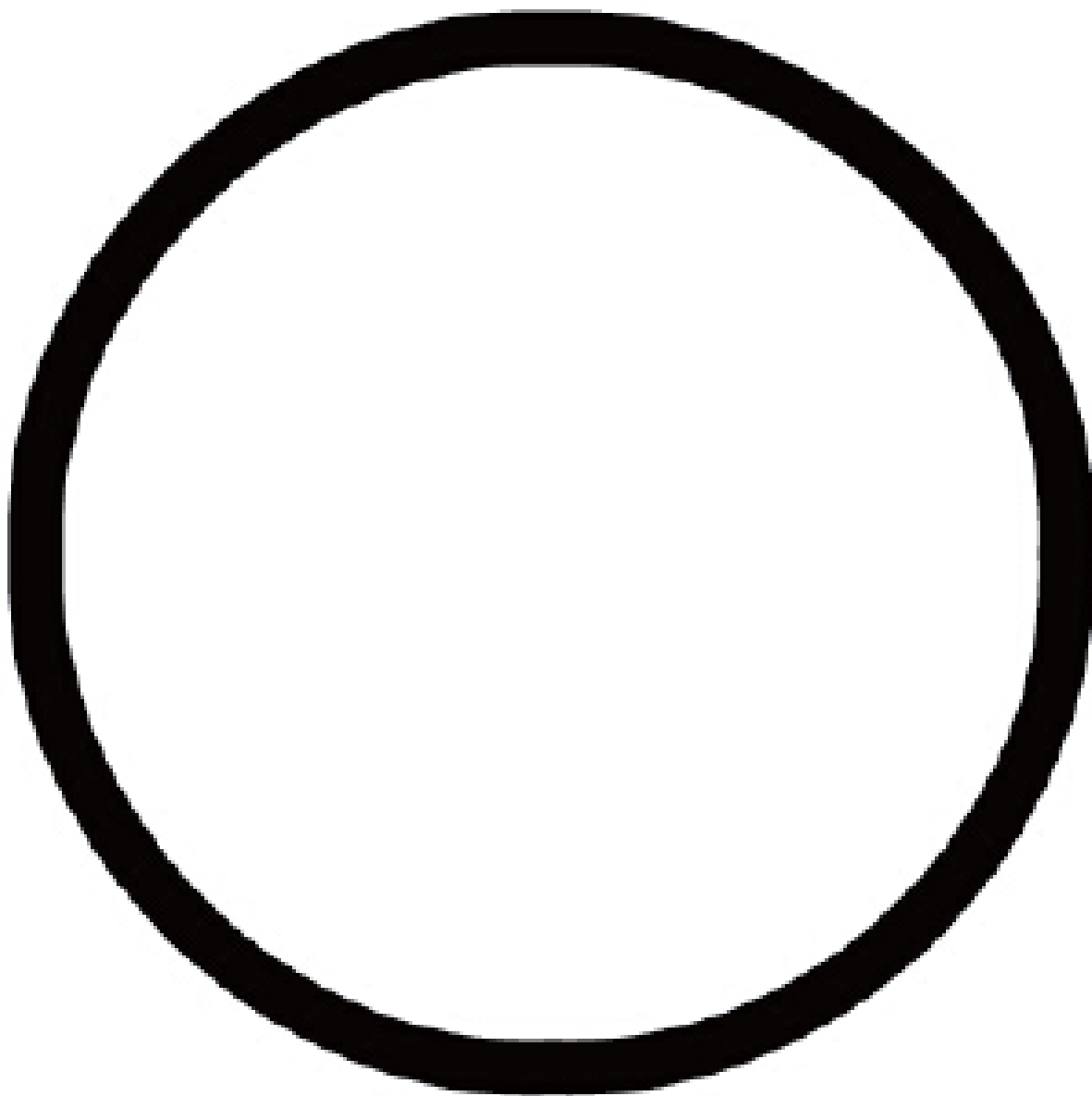


图4-1 启动事件的基本图形

所有的结束事件均以如图4-2所示的粗线圆形为基础。本节讲解的其他事件都是在其基础上“嵌入”对应类型的图形来表示一种结束事件的。

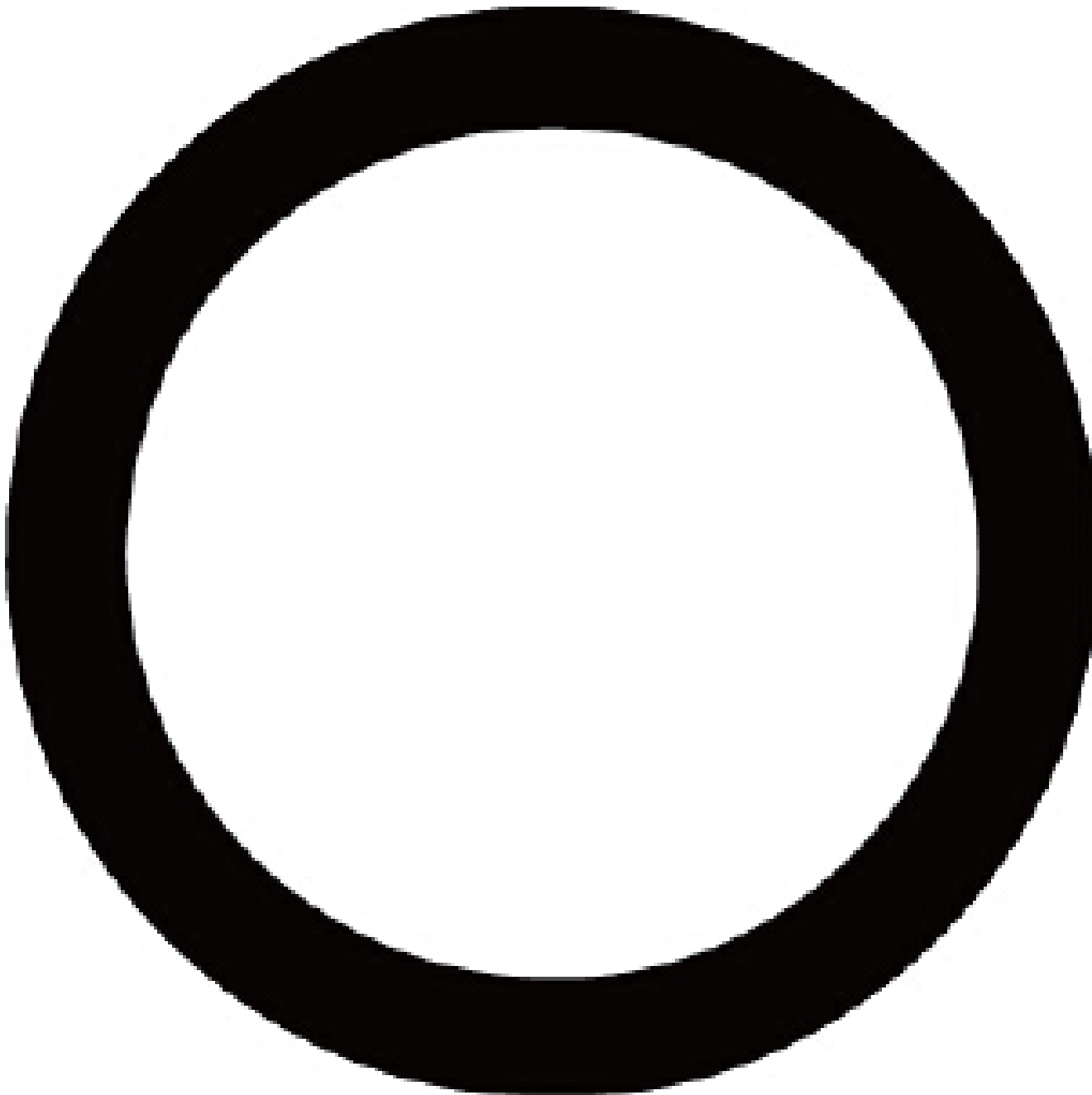


图4-2 结束事件的基本图形

4.1.1 启动事件

每个流程总是以启动事件作为入口（可以是不同类型的启动事件）。启动事件在BPMN 2.0规范中以一个细线圆形图形表示，如图4-1所示。启动事件又可以分为以下三种类型：

- 空启动事件
- 定时启动事件
- 异常启动事件

启动事件都是“捕获型”的，等待第三方触发后（读者可以联想一下观察者模式）才可以启动。在Activiti中可以通过调用API触发启动事件，在Hello World示例中就是通过调用runtimeService.startProcessInstance()方法触发来启动一个流程实例的。

1.空启动事件

图4-1展示了空启动事件的图形化表示，对应的XML描述如代码清单4-1所示。

代码清单4-1 空启动事件的XML描述

```
<startEvent id="startEvent" name="Start event" ></startEvent>
```

在代码清单4-1中表示一个启动事件，由于startEvent标签中没有任何其他的元素定义，所以称之为空启动事件，这也是其他启动事件的XML基本表示。

Activiti根据实际需求在空启动事件的基础上进行了扩展。表4-1列举了Activiti扩展的空启动事件属性。

表4-1 Activiti扩展的空启动事件属性

属 性 名 称	属 性 说 明	示 例
activiti:formKey	Activiti 扩展的 formKey 属性，可以用来指定空启动事件关联的表单文件。关于如何设置关联表单将在后续章节介绍	<startEvent id="startEvent" activiti:formKey="apply.form" />
activiti:initiator	Activiti 扩展的 initiator 属性，可以用来记录启动流程的人的 ID（也可以是用户的名称），启动流程之后此属性指定的变量就会自动设置当前人的名称	<startEvent id="startEvent" activiti:initiator="startUserId" /> 在流程流转中就可以通过获取变量 startUserId 得到启动时操作人 ID

除了以上属性之外，还可以在空启动事件上定义表单属性，同样在用户任务上也适用。启动事件可以配置相关表单，引擎提供了外置表单和动态表单两种表单形式。为了后面更好地理解这两种表单，先简单介绍这两种表单的区别。

· **外置表单**：表单的内容都存放在一个单独的“.form”文件中，可以是任何文本字符，一般用HTML书写，并且可以使用UEL^[1]占位符以达到动态填充的目的。启动事件中的activiti:formKey属性就是用来指定使用哪个表单文件（扩展名为.form）。

· **动态表单**：此类型表单不再使用activiti:formKey属性指定表单文件，而是在流程定义文件中设置表单中元素集合，包含各种输入框、下拉框等。对应的XML标示如下：

```
<activiti:formProperty id="name" name="Name" type="string"></activiti:formProperty>
```

activiti:formProperty可以有多个，对应一个表单中的多个字段（Field）。在实际应用中可以通过引擎提供的API读取、提交这些表单元素。

这里，对于不同表单的定义方式读者有一个概念即可，在“实战篇”会专门讲解详细使用方法及其适用的场景。

2.定时启动事件

定时启动事件可以用于一次性定时启动、特定时间间隔后启动，在BPMN 2.0规范中用一个形象的钟表图形表示，如图4-3所示。



图4-3 定时启动事件

定时启动事件常用于定期循环流程或一次性流程，例如公司每个月要产生一次业绩报表，可以设定每个月1号系统自动生成报表并自动启动公司的业务处理流程。

对应的，定时启动事件的XML描述和图形方式类似，在基本的启动事件中嵌套了一个定时事件定义timerEventDefinition，对应的XML描述如代码清单4-2所示。

代码清单4-2 定时启动事件的XML描述

```
<startEvent id="timerStartEvent" name="Timer start event for report process">
  <timerEventDefinition> #1
    <timeCycle>R1/2012-02-01T00:00/PM1M</timeCycle> #2
  </timerEventDefinition>
</startEvent>
```

代码清单4-2中#1处用timerEventDefinition描述一个定时事件，#2处的timerEvent-Definition表示一个定时事件，timeCycle使用ISO 8601^[2]标志定义了一个循环定时器，表示从2012年2月1日起每个月启动一次流程。可以通过多种方式设置定时事件，表4-2列举了定时启动事件的三种定时属性。

表4-2 定时启动事件的定时属性

属 性 名 称	属 性 说 明	示 例
timeDate	一次性定时启动，具体到一个日期，用 ISO 8601 格式设定	<timeDate>2012-12-12T00: 00: 00</timeDate> 在这个指定的日期启动一次流程
timeDuration	设置多长时间之后启动流程	<timeDuration>PT10M</ timeDuration > 部署流程或者输出流的上一个任务完成 10 分钟之后启动流程
timeCycle	周期性启动任务，用来设定循环的时间间隔，表示多长时间执行一次循环。同样还是用 ISO 8601 表示	<timeCycle>R3/PT10H</timeCycle> 循 环 3 次，每次间隔 10 小时。 在 timeDuration 的基础上添加了循环次数

3.异常启动事件

异常启动事件可以触发一个异常子流程，但是不能通过API（runtimeService.startProcess-Instance()方法）方式启动，它总是在另外一个流程抛出异常结束事件（4.1.2节）的时候被触发。

异常启动事件是“捕获型”，而异常结束事件是“抛出型”，和Java中的异常处理机制类似。异常结束事件对应throw，而异常启动事件是catch，并且执行catch后获取不同类型的异常。在BPMN 2.0规范中catch需要两者的错误信息匹配才能决定是否触发异常启动事件。

异常启动事件的图形化表示是在空启动事件基础上嵌套了一个雷电图号，如图4-4所示。

相对于空启动事件来说异常启动事件要复杂一些，必须嵌套在一个事件子流程（EventSubProcess，参见4.5.3）中，如图4-5中的虚线部分所示，代码清单4-3展示了一个简单的异常启动事件的应用。



图4-4 异常启动事件

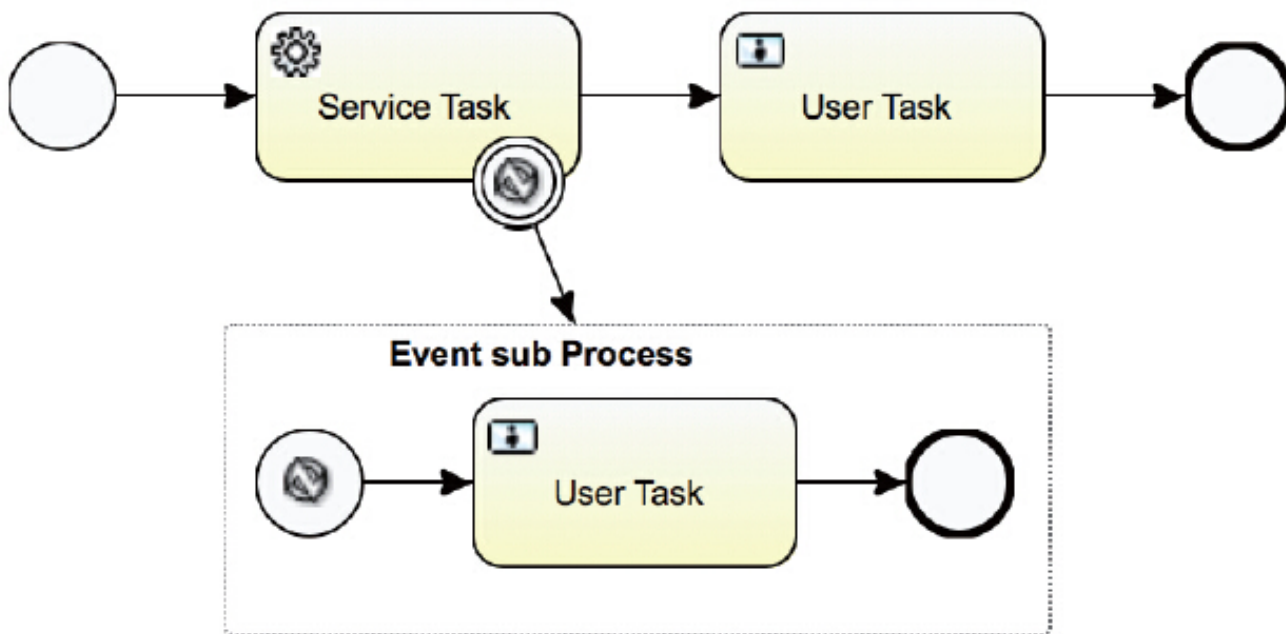


图4-5 EventSubProcess嵌套的异常启动事件

代码清单4-3 一个简单的异常启动事件的应用

```

<subProcess id="eventsubprocess1" name="Event sub Process" triggeredByEvent="true"> #1
  <startEvent id="errorstartevent1" name="Error start">
    <errorEventDefinition errorRef="AIA001"></errorEventDefinition> #2
  </startEvent>
  <scriptTask id="scripttask1" name="Script Task" scriptFormat="groovy"> #3
    <script><![CDATA[out.println "catch an error";]]></script>
  </scriptTask>
  <sequenceFlow id="flow3" name="" sourceRef="errorstartevent1" targetRef="scripttask1"></sequenceFlow>
  <endEvent id="endevent3" name="End"></endEvent>
  <sequenceFlow id="flow4" name="" sourceRef="scripttask1" targetRef="endevent3"> </sequenceFlow>
</subProcess>

```

代码清单4-3中#1处定义了一个事件子流程，表示的就是图4-5中的虚线部分。

#2处是被事件子流程包裹的异常事件，并且指定异常代码为“AIA001”，异常代码可以按照项目命名规范进行定义，例如Oracle的错误代码规范以ORA开头。在实际应用中根据业务类型和错误编号联合表示。

#3处定义了一个scriptTask用来输出“catch an error”文字，当捕获到异常代码为“AIA001”时触发异常启动事件。读者在了解异常启动事件时要结合异常结束事件来理解，两者的关系是相互依赖的。

4.消息启动事件

消息启动事件可以通过一个消息名称触发，从而启动一个流程实例；还可以结合消息抛出事件一起使用，由流程自动根据消息的名称启动对应的流程。借助这个功能在实际应用中可以为不同的业务处理结果启动不同的流程。

消息启动事件的图形化表示是在空启动事件的基础上嵌入一个空心的信封图标，如图4-6所示，对应的XML描述如代码清单4-4所示。



图4-6 消息启动事件

代码清单4-4 消息启动事件的XML描述

```
<message id="reSendFile" name="重新发送文件" />
...
<startEvent id="messageStart" >
  <messageEventDefinition messageRef="reSendFile" />
</startEvent>
```

可以通过Activiti提供的API触发消息启动事件，例如`runtimeService.startProcess-InstanceByMessage`（“重新发送文件”）可以启动一个包含消息名称为“重新发送文件”的流程，当然还可以使用像在第2章的Hello Word例子中启动流程的`runtimeService.startProcessInstanceByKey` (...) 和`runtimeService.startProcess-InstanceById` (...) 方法那样启动一个以消息启动事件为入口的流程，不过这些方法还有一些约束，在后续章节将会——通过实例方式进行比较说明。

4.1.2 结束事件

有始有终，结束事件和启动事件通常成对出现在流程定义中，当然在定义流程时也允许没有结束事件（但是开始事件不可缺少）。结束事件在BPMN 2.0规范中用一个圆形表示，和启动事件类似但有一点不同是线条加粗了，如图4-7所示。结束事件可以细分为以下几种类型：

- 空结束事件
- 异常结束事件
- 取消结束事件

结束事件表示流程（包含子流程）的结束，和启动事件的触发型不同的是结束事件总是**抛出型**的，也就是当流程执行到结束事件时会抛出一个执行结果。

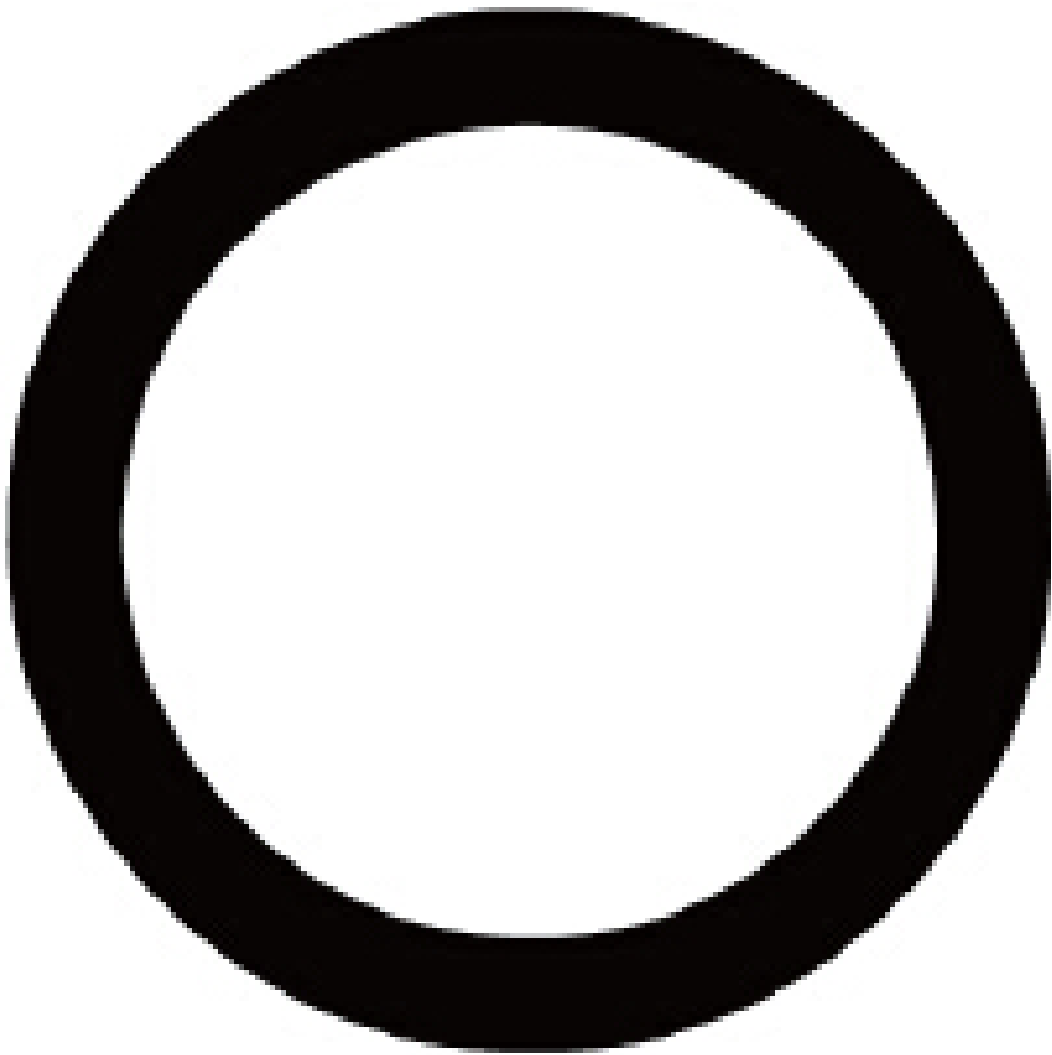


图4-7 结束事件的基本图形

所有的开始事件均以如图4-7所示的粗线条圆形为基础，本节讲解的其他事件都是在其基础上“嵌入”对应类型的图形来表示一种结束事件。

1.空结束事件

结束事件是抛出型的，空结束事件不处理抛出结果，也可以理解为抛出的是一个“空”，所以没有什么需要流程引擎处理的。对于空结束事件，正常结束后流程引擎就不会再执行其他的操作了，因为一切都已结束了，而且没有后续处理（结束事件不能有输出流）。空结束事件的图形表示如图4-7所示。

空结束事件和空启动事件的XML描述一样简单，如代码清单4-5所示。

代码清单4-5 空结束事件的XML描述

```
<endEvent id="end" name="my end event"/>
```

空结束事件一般用于正常结束流程，也就是说流程的执行过程一切都符合预设的业务逻辑，如果需要处理异常就需要使用异常结束事件或边界事件处理（4.6.1节）。

2.异常结束事件

异常结束事件相对于空结束事件来说就好理解一些了，空结束事件的抛出结果是“空”，而异常结束事件是有抛出结果的。它定义了需要抛出的错误代码，如果找到了异常开始事件定义的异常代码，则会触发异常开始事件，否则按照空结束事件规则处理。有一点需要注意，异常结束事件的错误代码不能为空。

异常结束事件的图形表示是在空结束事件的基础上添加了一个雷电符号，如图4-8所示。



图4-8 异常结束事件

异常结束事件的XML描述是在空结束事件中嵌套了一个异常事件errorEventDefinition，如代码清单4-6所示。

代码清单4-6 异常结束事件的XML描述

```
<endEvent id="endevent1" name="ErrorEnd"> #1
  <errorEventDefinition errorRef="AIA-001"></errorEventDefinition> #2
</endEvent>
```

代码清单4-6中的#1处定义了一个普通的结束事件。

#2处定义了一个errorEventDefinition表示一个异常事件，当遇到异常结束事件时，引擎查询定义为errorRef属性值的异常启动事件并触发。errorRef属性用来描述错误编号，可以按照项目命名规范进行定义，例如Oracle的错误代码规范以ORA开头。在实际应用中根据业务类型和错误编号联合表示。

 **注意** 不要把这里的异常和Java的异常混为一谈，BPMN 2.0中的异常仅仅描述异常定义及处理方式。

3.终止结束事件

终止结束事件是结束事件中的一个比较特别的类别，它可以终止一个流程实例的执行，例如在付费流程中因为某些原因导致流程不能继续运行，最终的结果是取消本次付费，所以需要提前结束流程实例的执行，此时可以把某个输出流指向到终止结束事件。空启动事件结束的仅仅是一条输出流，而终止结束事件结束的是整个流程实例。

终止结束事件的图形表示是在空结束事件的基础上添加了一个实心的圆点，如图4-9所示。

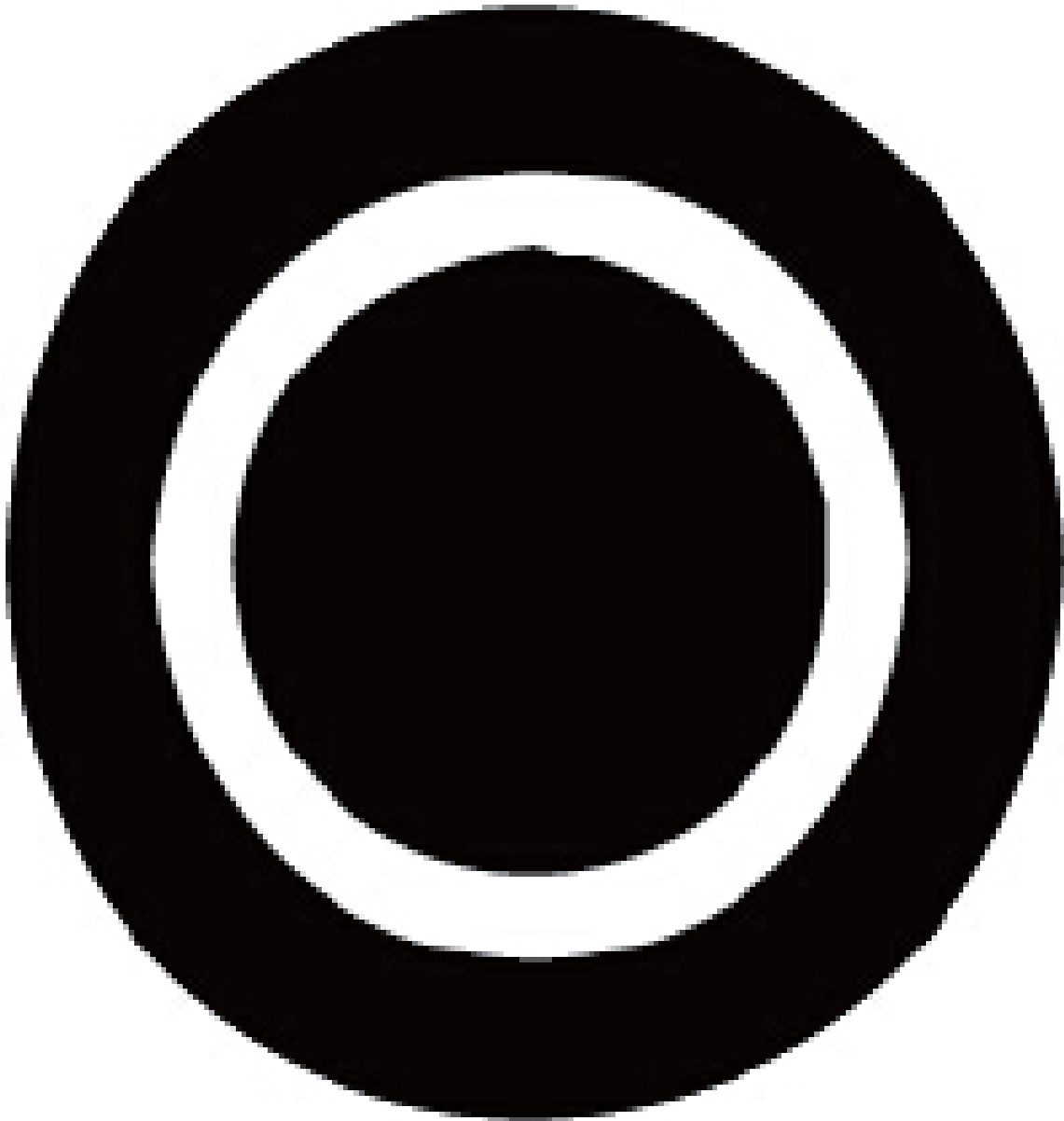


图4-9 异常结束事件

终止结束事件的XML描述是在空结束事件中嵌套了一个终止事件terminateEventDefinition，如代码清单4-7所示。

代码清单4-7 终止结束事件的XML描述

```
<endEvent id="terminateendevent1" name="TerminateEndEvent">
  <terminateEventDefinition></terminateEventDefinition>
</endEvent>
```

4.取消结束事件

取消结束事件可以取消一个[事务子流程](#)（4.5.4节）的执行，同时也只能在子流程中使用。当子流程执行过程中出现异常需要取消时，可以设置一个取消结束事件，当输出流指向到取消结束事件时流程将会中断执行。取消结束事件还可以和取消边界事件（4.6.1节）配合使用针对取消操作做后续处理。

取消结束事件的图形表示是在空结束事件基础上嵌入一个“X”图形，如图4-10所示，对应的XML描述如代码清单4-8所示。

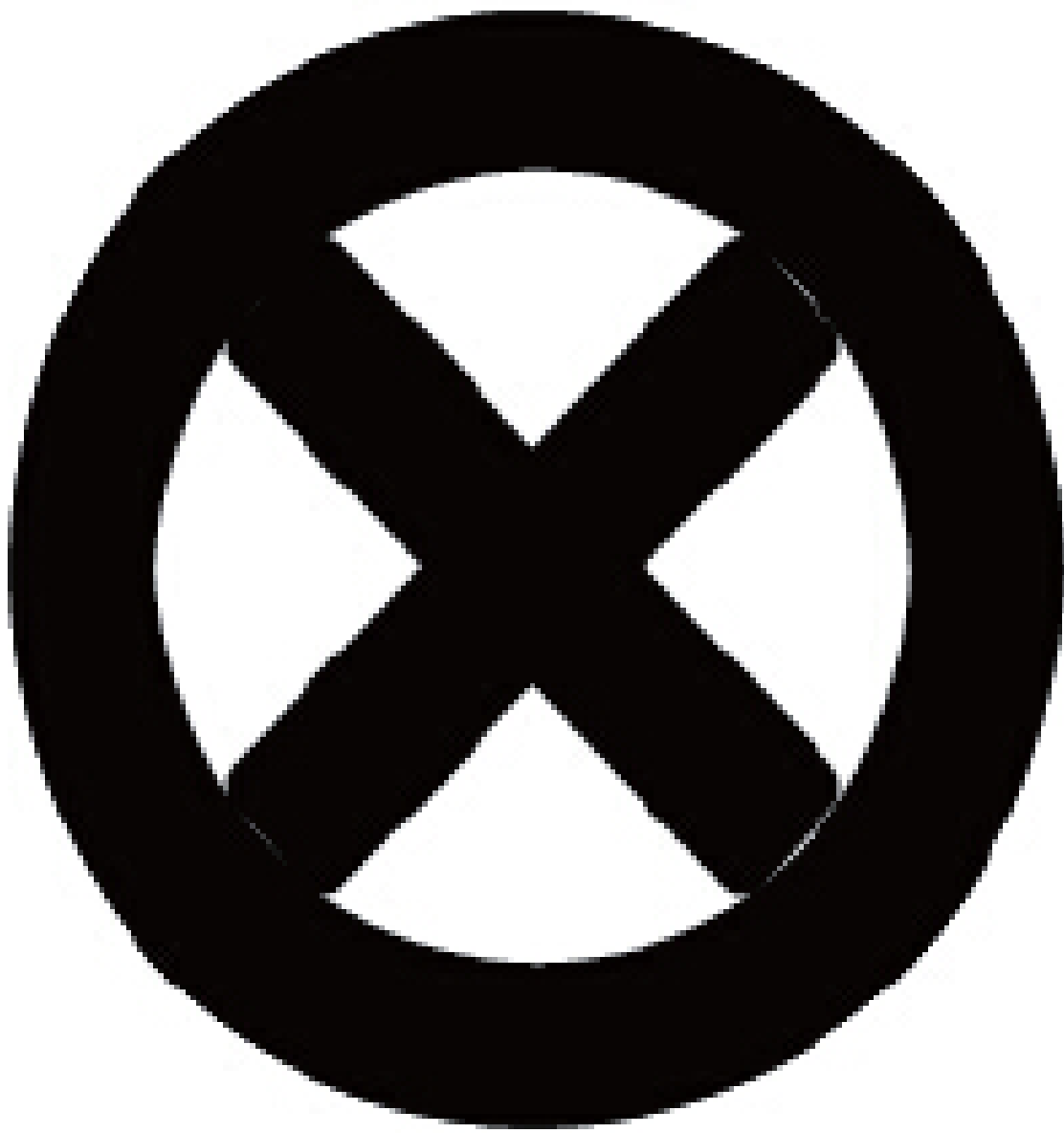


图4-10 取消结束事件

代码清单4-8 取消结束事件的XML描述

```
<endEvent id="myCancelEndEvent">
  <cancelEventDefinition />      #1
</endEvent>
```

代码清单4-8的#1处定义了cancelEventDefinition标签来声明这是一个取消事件，读者可结合事务子流程（4.5.4节）中的取消结束事件来理解。

[1] 关于UEL读者可以参考JAVAE6中的定义：<http://docs.oracle.com/javacc/6/tutorial/doc/gjddd.html>。

[2] http://en.wikipedia.org/wiki/ISO_8601

4.2 顺序流

顺序流是两个模型之间的连接者，可以把顺序流比作人体的动脉，每一条连接到不同的器官，在BPMN 2.0规范中每个输出流连接到不同的活动、事件。如果一个元素在流程执行期间被访问，流程会沿着该元素所有输出顺序流继续执行。这意味着BPMN 2.0默认为并行：多个输出顺序流会创建多条独立、并行的执行路径。

顺序流可以细分为两种：

- 标准顺序流
- 条件顺序流

4.2.1 标准顺序流

标准顺序流可以用来连接两个或多个模型建立关系，在BPMN2.0规范中如图4-11所示，对应的XML描述如代码清单4-9所示。



图4-11 异常结束时间

代码清单4-9 标准顺序流的XML描述

```
<sequenceFlow id="flow" sourceRef="startEvent" targetRef="userTask"></sequenceFlow>
```

从代码清单4-8中可以看出，用sequenceFlow表示一个顺序流，用sourceRef属性指定顺序流的源，用targetRef属性指定顺序流的目的模型。

Activiti还对顺序流进行了扩展，允许开发人员在顺序流上添加监听器（参考4.7.1节）。

4.2.2 条件顺序流

条件顺序流是在标准顺序流上添加了条件表达式，只有满足条件才能通过顺序流到达目标活动。BPMN 2.0规范定义的条件顺序流图形如图4-12所示。



图4-12 BPMN 2.0规范的条件顺序流图形

 **说明** 在Activiti Designer中设置条件顺序流没有图4-12所示的图形，取而代之的是标准顺序流，在设计的时候不设置condition的就是标准顺序流，设置了condition的就变成了条件顺序流。

条件顺序流的XML描述是在顺序流中添加了条件表达式标签conditionExpression，并且在conditionExpression中设置UEL^[1]表达式用于计算逻辑值，如代码清单4-10所示。

代码清单4-10 条件顺序流的XML描述

```
<sequenceFlow id="flow5" name="" sourceRef="startevent2" targetRef="usertask1">
  <conditionExpression xsi:type="tFormalExpression" #1
    <![CDATA[{pass == true}]]> #2
  </conditionExpression>
</sequenceFlow>
```

代码清单4-9中的#1处使用conditionExpression标签定义了一个条件表达式，并且使用xsi:type指定表达式的类型为tFormalExpression，目前条件表达式仅支持UEL。#2处的条件使用CDATA方式描述，在运行时交由引擎根据变量pass的值和true进行比较，还可以简化为<![CDATA[{pass}]]>。

[1] <http://uel.java.net/>

4.3 任务

任务是一个流程中最重要的组成部分，根据业务需求的不同也分为很多种类型：

- 用户任务
- 脚本任务
- WebService任务
- 业务规则任务
- 邮件任务
- Mule任务
- Camel任务
- 手动任务
- Java service任务
- Shell任务

所有的任务的图形化表示都是以一个矩形为基础，在左上角添加具体类型的图标。大多数任务都有单独的XML定义，用来描述一种特定任务类型，也有部分任务是基于serviceTask扩展的。

4.3.1 用户任务

顾名思义，用户任务需要人来参与，因为它必须被人为触发（完成任务动作）。用户任务可以定义任务的名称（例如“领导审批”）、优先级、到期日和任务处理人（可以是人、组或者两者的组合）。图4-13展示了用户任务的图形。

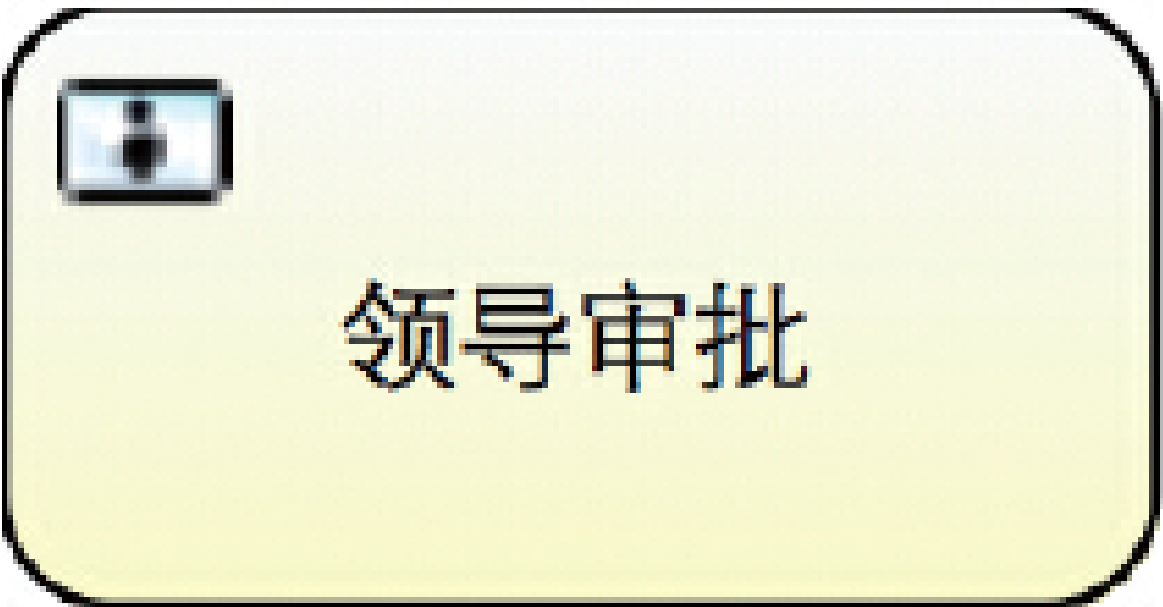


图 4-13 用户任务

用户任务使用userTask表示，代码清单4-11中定义了一个基本的用户任务，把任务分配给ID为henryyan的用户办理。

代码清单4-11 用户任务的XML描述——分配任务给henryyan

```
<userTask id="leaderAudit" name="领导审批"> #1
  <humanPerformer> #2
    <resourceAssignmentExpression>
      <formalExpression>henryyan</formalExpression> #3
    </resourceAssignmentExpression>
  </humanPerformer>
</userTask>
```

代码4-11中#1处定义了一个名称为“领导审批”用户任务，#2处用humanPerformer标签定义表示把这个任务分配给一个人，并且在#3处用一个表达式指定了用户任务分配给用户henryhan办理。

用户任务除了可以分配给一个用户之外还可以分配给组或两者混合，代码清单4-12展示了把一个任务同时分配给多个候选人。

代码清单4-12 用户任务的XML描述——分配任务给多个候选人

```
<userTask id="leaderAudit" name="领导审批">
  <potentialOwner> #1
    <resourceAssignmentExpression>
      <formalExpression>
        user(henryyan),group(leader),manager #2
      </formalExpression>
    </resourceAssignmentExpression>
  </potentialOwner>
</userTask>
```

代码清单4-11中的#1处用potentialOwner标签来描述一个潜在的用户、组集合，#2处把“领导审批”这个用户任务同时分配给用户henryyan、组leader和组manager。现在读者要问为什么group(leave)和manager都是分配到组呢？这是因为如果不添加user或group关键字说明括号中的候选人属于什么类型，那么BPMN 2.0规范默认为组（group）。

此外Activiti在BPMN 2.0的基础上进行扩展，可以简化设置用户、组的方式，而且支持动态（运行时）获取用户、组分配给用户任务；还可以为用户任务设置创建、分配、完成监听。表4-3列举了Activiti扩展的用户任务属性。

表4-3 Activiti扩展的用户任务属性

属 性 名 称	属 性 说 明	示 例
activiti:assignee	用来指定用户任务的处理人，代替了 humanPerformer 的功能	<userTask id="leaderAudit" name=" 领导审批 " activiti:assignee="henryyan"/>
activiti:cadidateUsers	用来指定用户任务的候选人，多个用逗号分开，代替了 potentialOwner 的功能	<userTask id="leaderAudit" name=" 领导审批 " activiti:cadidateUsers="henryyan" />
activiti:cadidateGroups	用来指定候选组，多个用逗号分开，代替了 potentialOwner 的功能	<userTask id="leaderAudit" name=" 领导审批 " activiti:cadidateGroups="leader, manager" >
activiti:dueDate	设置用户任务到期日，通常用变量代替而不是指定一个具体的日期，因为这样毫无意义	<userTask id="leaderAudit" name=" 领导审批 " activiti:dueDate="\${overDate}" >
activiti:priority	用户任务的优先级，取值区间 [0, 100]	<userTask id="leaderAudit" name=" 领导审批 " activiti:priority="\${priority}" >

代码清单4-13展示了通过Activiti的扩展属性设置用户任务的办理人，以代替BPMN 2.0规范中繁琐的配置。

代码清单4-13 使用Activiti扩展属性设置用户任务的办理人和候选人

```
<userTask id=" leaderAudit " name="领导审批"  
activiti:candidateUsers=" henryyan"  
activiti:candidateGroups="group1, group2">  
</userTask>
```

除了表4-3中的属性之外，Activiti还允许在用户任务上添加任务监听，监听的选项有：create（创建任务）、assignment（分配任务）、complete（完成任务），如代码清单4-14所示。

代码清单4-14 使用Activiti扩展标签设置用户任务的办理人和候选人

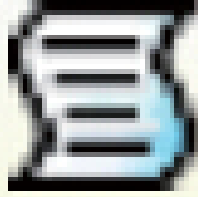
```
<userTask id="leaderAudit" name="领导审批">  
  <extensionElements>  
    <activiti:taskListener class="me.kafeitu.activiti.LeaderTaskListener" event="complete" />  
  </extensionElements>  
</userTask>
```

代码清单4-13中的配置可以在“领导审批”用户任务完成的时候触发Leader-TaskListener监听。除了可以通过class属性指定一个Java的class文件（jar包中的全路径）之外，还可以定义expression和delegateExpression以表达式方式处理监听。

关于监听器的设置在4.7节中会详细讲解。

4.3.2 脚本任务

脚本任务可以运行引擎依赖的语言之外的脚本语言，例如Activiti支持的Groovy、Javascript、Juel。在BPMN 2.0规范中脚本任务的图形表示如图4-14所示。



Script Task

图4-14 脚本任务

值得注意的是，脚本任务的代码需要符合JSR-223^[1] (Java平台脚本，scripting for the java platform) 规范。

代码清单4-15中定义了一个Groovy类型的脚本任务。

代码清单4-15 使用Groovy作为脚本引擎的脚本任务

```
<scriptTask id="initvars" name="初始化变量" #1
  scriptFormat="groovy"> #2
  <script><![CDATA[ #3-S
    def name = "HenryYan";
    execution.setVariable('name', name);
  ]]> #3-E
</scriptTask>
<scriptTask id="printvars" name="输出变量" #4
  scriptFormat="groovy"> #5
  <script><![CDATA[out:println name;]]></script> #6
</scriptTask>
```

代码清单4-15中定义了两个Groovy脚本任务，其中#1、#4处分别使用`scriptTask`定义了两个脚本任务，#2、#5处通过`scriptFormat`指定脚本任务为Groovy类型，#3、#6处是用来设置脚本任务执行的代码。第一个脚本任务用来设置变量name的值为HenryYan，第二个脚本任务用来输出变量name。

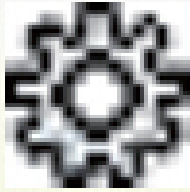
Activiti除了默认支持的三种脚本语言Groovy、JavaScript、Juel，还允许使用其他的脚本语言，前提是把依赖的jar包设置到classpath中。表4-4列举了脚本任务的属性。

表4-4 脚本任务的属性

属性名称	属性说明	示例
scriptFormat	用来指定符合 JSR-223 规范的脚本语言的类型	<code><scriptTask id="scripttask1" name=" 初始化变量 " scriptFormat="groovy"/></code>
activiti:resultvariable	Activiti 在原 BPMN 2.0 规范中的脚本任务基础上进行了扩展，可以把脚本处理的结果保存到一个变量中	<code><scriptTask id="scripttask1" name=" 初始化变量 " scriptFormat="juel" activiti:resultVariable="name"/></code> 注意 <code>activiti:resultVariable</code> 指定的变量名称需要在脚本中预先定义才能使用

4.3.3 Java Service任务

首先声明Java Service不属于BPMN 2.0规范，而是Activiti扩展的专门应用于Java语言的serviceTask。其图形表示如图4-15所示。



My Java Service Task

图4-15 Java Service任务

Java Service任务允许定义一个实现了指定接口的Java类，或者执行一个表达式；还可以像脚本任务一样把结果保存到一个变量中。

Java Service任务对应的XML如代码清单4-16所示。

代码清单4-16 Java Service任务对应的XML描述

```
<serviceTask id="myServiceTask" name="Learn Java Service Task" #1
  activiti:class="me.kafeitu.activiti.JavaServiceDelegate" > #2
</serviceTask>
```

代码清单4-16中的#1处使用serviceTask标签定义一个Java Service任务，#2处设置activiti:class= "me.kafeitu.activiti.JavaServiceDelegate" 表示这个任务需要执行一个Java类，这个Java类需要遵循Activiti的Java Service规范实现JavaDelegate、ActivityBehavior中的一个接口。

在指定一个Java类的同时还可以配置执行Service时传入的变量，这样在执行Java类的时候可以读取预先设置的变量值。

除了使用activiti:class指定一个Java类之外，还可以使用其他的几种方式定义Java Service任务需要执行的Java对象，表4-5列举了Activiti扩展的Java Service任务的属性。

表4-5 Activiti扩展的Java Service任务的属性

属性名称	属性说明	示例
activiti:class	实现了接口 JavaDelegate 或 ActivityBehavior 的 Java 类	<pre><serviceTask id="serviceTask1" activiti:class=" me.kafeitu.activiti.JavaServiceDelegate "/></pre>
activiti:expression	可以使用 UEL 定义需要执行的任务内容，例如计算公式、调用 Bean 对象的方法；并且在执行任务的时候可以使用流程变量作为参数。Bean 实例除了可以使用 new 创建之外，还可以使用 Spring 代理	<pre>< serviceTask id=" serviceTask1" activiti:expression= "#{leaveService.back()}" /></pre> 其中 leaveService 作为一个流程变量存在，和 activiti:class 一样需要实现 JavaDelegate 接口或 ActivityBehavior 接口，另外还实现了 java.io.Serializable 接口以便引擎可以序列化变量
activiti:delegateExpression	功能和 activiti:class 类似，而且同样需要实现 JavaDelegate 或 ActivityBehavior 中一个接口，只不过这里不是指定一个具体的实现类，而是在运行时动态设置。和 activiti:expression 类似，Bean 可以使用 new 创建也可以用 spring 代理	<pre>< serviceTask id=" serviceTask1" activiti:expression= "\${leaveBackDelegate}" /></pre> 执行到此任务时引擎会从变量中查询名称为 leaveBackDelegate 的 Bean 对象，然后调用实现接口的方法

属性名称	属性说明	示例
activiti:resultVariable	此属性仅适用于 activiti:expression 类型的 Java Service，可以把一个表达式的执行结果保存到 activiti:resultVariable 指定的变量名称中	<serviceTask id= " serviceTask1" activiti:expression="{leaveService.back()}"activiti:resultVariable="backDate"/> 把 leaveService.back() 执行的结果保存到变量 backDate 中

4.3.4 Web Service任务

通过Web Service任务可以调用外部的Web Service资源，完成调用只需要一些必须的配置就可以，并且支持标准的Web Service和REST风格的Service。Web Service任务的图形表示如图4-16所示。



图4-16 Web Service 任务

Web Service任务对应的XML描述如代码清单4-17所示。

代码清单4-17 Web Service任务的XML描述

```
<serviceTask id="webServiceTask" implementation="##WebService"> #1
<ioSpecification> #2
  <dataInput itemSubjectRef="tns:prettyPrintCountRequestItem"
    id="dataInputOfServiceTask"/>
  <dataOutput itemSubjectRef="tns:prettyPrintCountResponseItem"
    id="dataOutputOfServiceTask"/>
  <inputSet>
    <dataInputRefs>dataInputOfServiceTask</dataInputRefs>
  </inputSet>
  <outputSet>
    <dataOutputRefs>
      dataOutputOfServiceTask
    </dataOutputRefs>
  </outputSet>
</ioSpecification>
<dataInputAssociation>
  <sourceRef>PrefixVariable</sourceRef>
  <targetRef>prefix</targetRef>
</dataInputAssociation>
<dataInputAssociation>
  <sourceRef>SuffixVariable</sourceRef>
  <targetRef>suffix</targetRef>
</dataInputAssociation>
<dataOutputAssociation>
  <sourceRef>prettyPrint</sourceRef>
  <targetRef>OutputVariable</targetRef>
</dataOutputAssociation>
</serviceTask>
```

在代码清单4-17中，#1处使用implementation= “##WebService” 属性设置Task为Web Service类型，#2处通过ioSpecification元素定义输入、输出参数：在元素dataInputAssociation中定义了数据输入关系；在元素dataOutputAssociation中定义了数据输出关系。

4.3.5 业务规则任务

在企业应用中一般都会使用可维护的规则库来管理复杂多变的业务规则，可以把业务逻辑和规则分开维护，一旦规则有变动，只需要更改预设规则即可。业务规则任务可以根据流程变量的值处理预设的业务规则。

Activiti对业务规则提供了很好的支持，目前支持比较流行的JBoss规则引擎——Drools^[2]。只需把含有业务规则任务的流程文件和规则引擎文件 “.drl” 一同打包部署到系统中，同时把Drools的jar包添

加到classpath即可实现Activiti驱动规则引擎。业务规则任务的图形表示如图4-17所示。

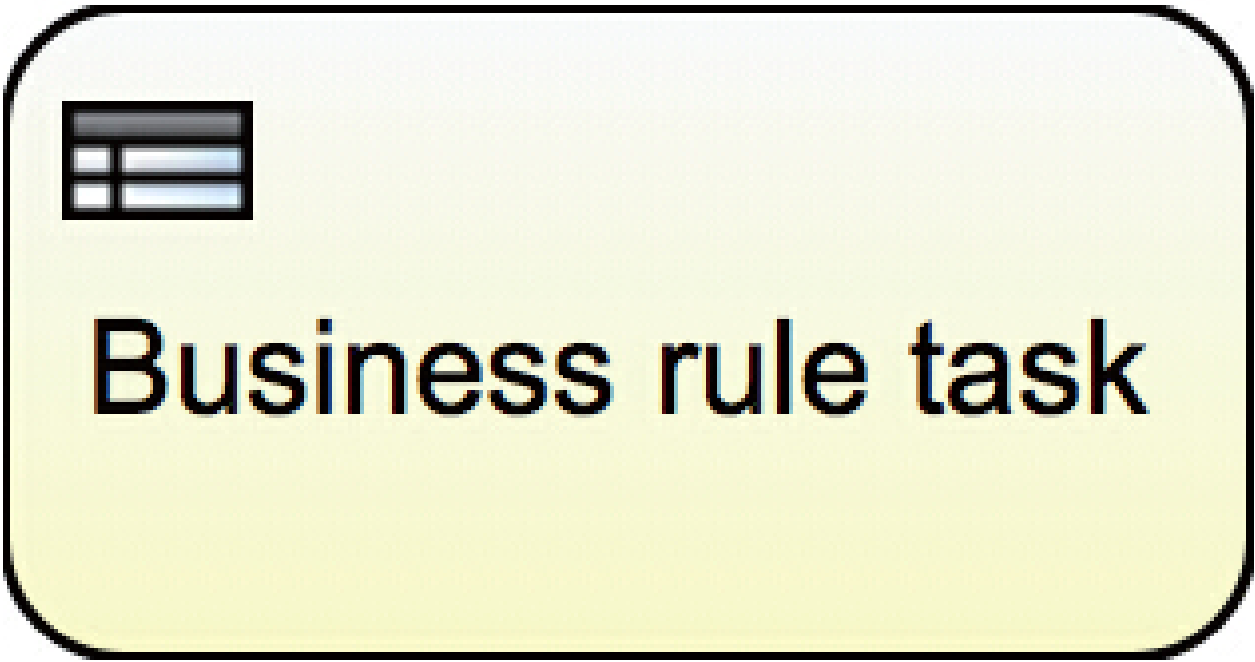


图4-17 业务规则任务

业务规则任务的XML描述如代码清单4-18所示。

代码清单4-18 业务规则业务规则任务的XML描述

```
<businessRuleTask id="businessruleruletask1" name="Business rule task"
  activiti:rules="rule1, rule2" activiti:ruleVariablesInput="{message}"
  activiti:exclude="false" activiti:resultVariableName="rulesOutput">
</businessRuleTask>
```

在介绍Activiti的扩展属性之前，读者需要先明白规则引擎的作用，简单来说就是把业务数据交由规则引擎处理，规则引擎根据不同业务规则（各种条件的判断）计算得出最终结果，最后把结果返回给调用者，在BPMN 2.0中的调用者为流程引擎（在本书中为Activiti）。

Activiti在原有的业务规则规范上扩展了一些属性以便与Drools整合，表4-6列举了Activiti扩展的业务规则任务的属性。

表4-6 Activiti扩展的业务规则任务的属性

属 性 名 称	属 性 说 明	示 例
activiti:rules	在规则文件 .drl 中定义的规则名称，多个规则用逗号分隔。要执行规则文件中的全部规则，将该属性设置为空即可	<businessRuleTask id="businessruleruletask1" activiti:rules="rule1, rule2" />

(续)

属性名称	属性说明	示例
activiti:ruleVariablesInput	业务规则执行需要的数据源，使用 <code>\${fooVar}</code> 方式定义，多个规则用逗号分隔	<pre><businessRuleTask id="businessruleruletask1" activiti:ruleVariablesInput="\${message}" /></pre>
activiti:resultVariableName	规则执行结果变量，变量的值为 <code>ruleVariablesInput</code> 定义的变量集合 (ArrayList)	<pre><businessRuleTask id="businessruleruletask1" activiti:resultVariableName=" rulesOutputVariableList" /></pre>
activiti:exclude	用来设置是否排除某些规则 (rule)，如果值为 <code>false</code> ，不排除按照 <code>activiti:rules</code> 规则执行；如果设置为 <code>true</code> ，则忽略 <code>activiti:rules</code> 指定的规则。如果设置为 <code>false</code> 的同时 <code>activiti:rules</code> 值为空，则不执行任何规则	<pre><businessRuleTask id="businessruleruletask1" activiti:rules="rule2"activiti:exclude="true" /></pre> 此例中忽略规则 rule2，仅执行规则 rule1

4.3.6 邮件任务

邮件任务可以通过Activiti发送邮件，其中的邮件信息通过变量方式传递。邮件任务不属于BPMN 2.0规范而是和Java Service任务类似，由Activiti扩展而来专门用于处理邮件任务，其图形表示如图4-18所示。



图4-18 邮件任务

邮件任务是在serviceTask的基础上由Activiti添加了扩展属性`activiti:type="mail"`实现的。邮件任务的XML描述如代码清单4-19所示。

代码清单4-19 邮件任务的XML描述

```
<serviceTask id="mailtask1" name="Mail Task" activiti:type="mail">
  <extensionElements>
    <activiti:field name="to" expression="henryyan@gmail.com"></activiti:field>
    <activiti:field name="from" expression="yanhonglei@gmail.com"></activiti:field>
    <activiti:field name="subject" expression="hello henryyan"></activiti:field>
    <activiti:field name="charset" expression="UTF-8"></activiti:field>
    <activiti:field name="html">
      <activiti:expression><![CDATA[你好，我是${userName}。]]></activiti:expression>
    </activiti:field>
  </extensionElements>
```

发送邮件需要配置邮件服务器信息到流程引擎中，可以在activiti.cfg.xml定义的引擎属性中设置。表4-7列举了配置邮件服务器的属性。

表4-7 配置邮件服务器的属性

属 性 名 称	是 否 必 须	描 述
mailServerHost	否	邮件服务器的主机名（例如 smtp.gmail.com），默认为 localhost
mailServerPort	是，如果不是默认的端口	SMTP 通信端口，默认为 25；如果使用 SSL，则为 465
mailServerDefaultFrom	否	发件人 email，如果不设置，默认为 acitiviti@activiti.org
mailServerUsername	否，根据服务器是否需要认证设置	邮件服务认证账号，默认为空
mailServerPassword	否，根据服务器是否需要认证设置	邮件服务认证密码，默认为空

代码清单4-19中的邮件任务配置统一使用activiti:field元素来配置，name不同表示不同的参数，均支持变量方式指定值。表4-8列举了邮件任务的属性。

表4-8 邮件任务的属性说明

属 性 名 称	属 性 说 明	示 例
to	必填。收件人，多个收件人由逗号分隔的列表来定义	<pre><activiti:field name="to" expression="\${to}" /></pre>
from	邮件发送人地址。如果不设置，使用引擎的 mailServerDefaultFrom 属性指定的发件人	<pre><activiti:field name="from" expression="\${from}" /></pre>
subject	发件人 email，如果不设置，默认为 acitiviti@activiti.org	<pre><activiti:field name="subject" expression="\${subject}" /></pre>
cc	抄送人列表	<pre><activiti:field name="cc" expression="\${cc}" /></pre>
bcc	密送人列表	<pre><activiti:field name="cc" expression="\${cc}" /></pre>
charset	邮件内容字符集，建议使用 UTF-8 防止中文乱码	<pre><activiti:field name="charset" expression="\${charset}" /></pre>

属 性 名 称	属 性 说 明	示 例
text	纯文本格式的邮件内容	<pre><activiti:field name="text" expression="\${email}" /></pre>
html	html 格式的邮件内容	<pre><activiti:field name="html"> <activiti:expression> <![CDATA[<html> <body> 你好，我是 \${userName}。 </body> </html>]]> </activiti:expression> </activiti:field></pre>

4.3.7 Camel任务

Camel^[3]是由Apache开源组织开发的基于Apache 2协议的企业系统整合框架，是EIP（Enterprise Integration Patterns，企业集成模式）的实现，是用来解决消息路由的框架。类似的产品还有Mule、Spring Intergration等。

Camel任务是使用Java语言编写的，所以不包含在BPMN 2.0规范中，而是由Activiti在serviceTask的基础上扩展的一个任务模型。Activiti对其提供了很好的集成支持。可以通过activiti:delegateExpression属性指定Camel的上下文，在运行时由引擎调用Camel路由处理任务。Camel任务的图形表示如图4-19所示。

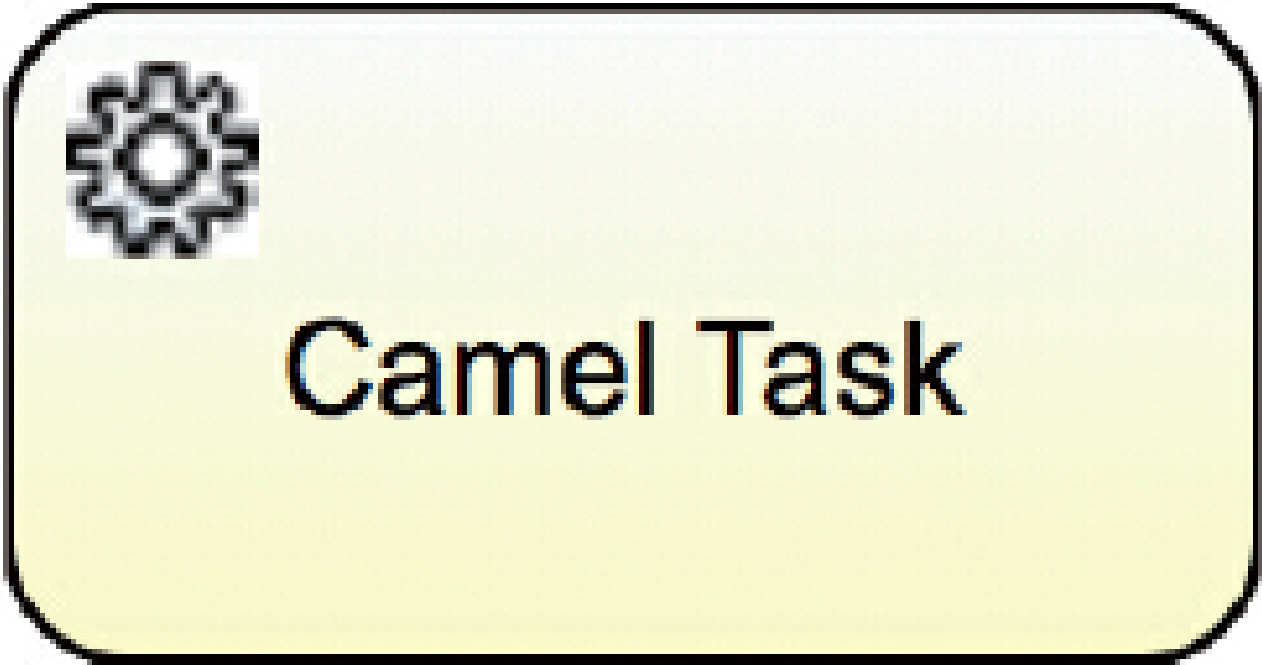


图4-19 Camel任务

Camel任务和邮件任务的XML类似，都是在serviceTask的基础上由Activiti扩展而来，代码清单4-20列出了Camel的XML描述。

代码清单4-20 Camel任务的XML描述

```
<serviceTask id= "camelTask1" name="Camel Task" activiti:delegateExpression="${camel}" />
```

在代码清单4-20中定义了一个Camel任务，通过属性activiti:delegateExpression指定一个Camel的上下文对象，\${camel}表示由Activiti调用名称为camel的Bean对象处理Camel任务，它可以定义在Activiti配置文件中或由Spring代理。

4.3.8 Mule任务

Mule任务^[4]是由MuleSoft开发的一款轻量级开源ESB（Enterprise System Bus，企业系统总线）产品，和Camel任务功能类似，但是它们是基于不同的标准实现的，Camel任务是基于传统EIP实现的。在使用方式上Camel任务使用链式编程配置路由，而Mule使用非侵入（或者很少侵入）方式，通过可视化的IDE工具使用流程图模式配置路由。

Mule任务和Camel任务一样，也不是BPMN 2.0规范的一部分，Activiti针对Mule任务进行了整合，可以直接在流程中通过添加Task的方式调用Mule处理业务逻辑。

Activiti对Camel的扩展使用的是serviceTask，而对于Mule任务使用另外一种发送任务——sendTask。

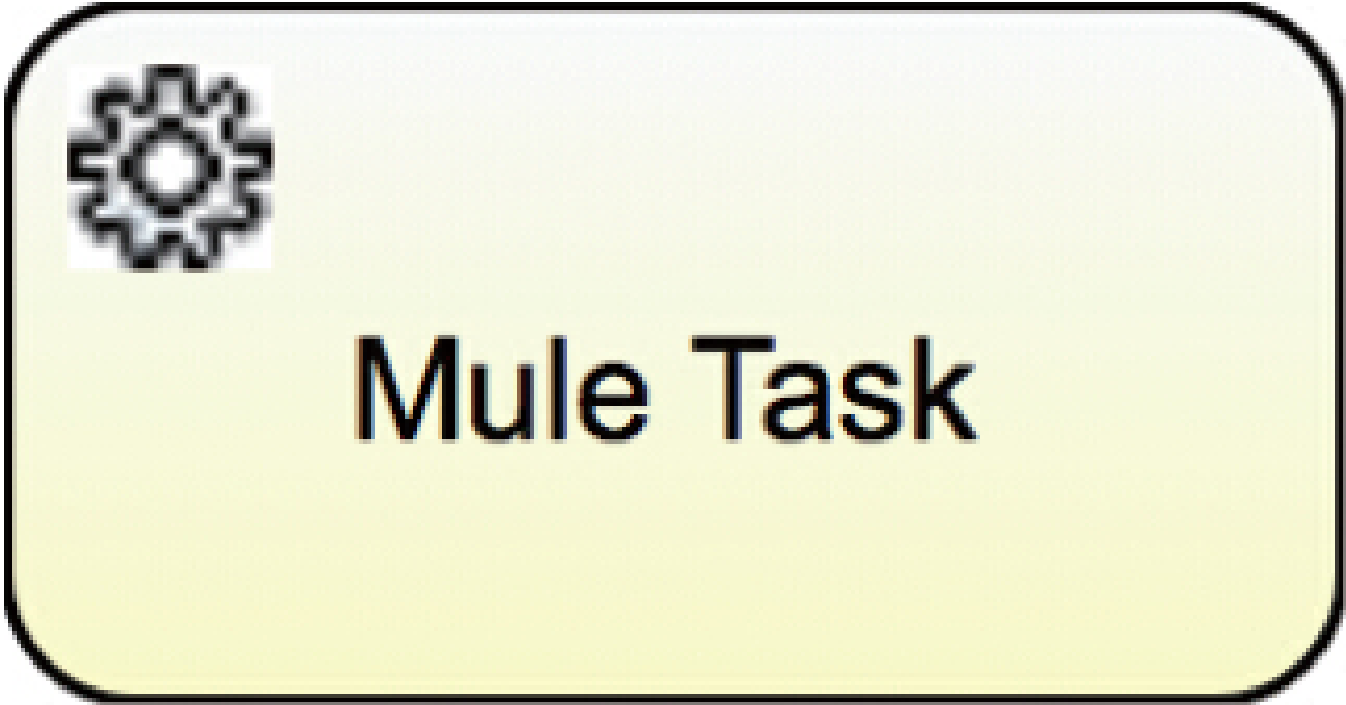


图4-20 Mule任务

Mule任务的图形化表示使用一个黑色填充的信封表示，如图4-20所示。代码清单4-21列出了Mule任务的XML描述。

代码清单4-21 Mule任务的XML描述

```
<sendTask id="muleTask" activiti:type="mule"> #1
  <extensionElements>
    <activiti:fieldname="endpointUrl"> #2
      <activiti:string>vm://in</activiti:string>
    </activiti:field>
    <activiti:fieldname="language"> #3
      <activiti:string>juel</activiti:string>
    </activiti:field>
    <activiti:fieldname="payloadExpression"> #4
      <activiti:string>"hi"</activiti:string>
    </activiti:field>
    <activiti:fieldname="resultVariable"> #5
      <activiti:string>resultVar</activiti:string>
    </activiti:field>
  </extensionElements>
</sendTask>
```

- 代码清单4-21中的#1处定义activiti:type= “mule” 来指定任务的类型，在extension-Elements标签中配置一系列调用Mule的参数。
- 代码清单4-21中的#2处字段endpointUrl指定了协议类型为vm，类似Java虚拟机，还可以指定其他的协议调用Mule，例如JMS或其他的远程访问协议。
- 代码清单4-21中的#3处字段language指定了调用payloadExpression的语言，此处指定为JUEL。在实际使用时可以通过Activiti的任务变量动态设置。
- 代码清单4-21中的#4处字段payloadExpression用来指定Mule消息的payload名称。
- 代码清单4-21中的#5处字段resultVariable用来保存Mule任务的执行结果，可以通过Activiti API获取。

4.3.9 手动任务

手动任务是比较特殊的任务之一，它不做任何事情，仅用来定义BPM不能完成的任务，需要人工参与建模，流程引擎无需关心如何处理它，因为根本不需要处理，所以Activiti把手动任务当做一个空任务来处理，当到达此任务时由引擎自动完成并转向到下一个任务。



Manual Task

图4-21 手动任务

手动任务的图形用一个手的图标表示，如图4-21所示。代码清单4-22列出了手动任务的XML描述。

代码清单4-22 手动任务的XML描述

```
<manualTask id="manualTask" name="Manual task" />
```

4.3.10 接收任务

接收任务是一个功能简单且单一的任务，在任务创建之后开始等待消息的到来，直到被触发才会完成任务。Activiti实际上把接收任务作为一个Java类型的接受任务，仅能通过[RuntimeService](#)接口的[signal\(\)](#)方法发送信号（signal）触发接收任务。原理类似线程的等待和恢复执行，只不过线程是在内存中操作，而接受任务的状态保存在数据库中，在调用Activiti的API触发流程的接收任务后，引擎把当前流程由等待状态恢复为可执行状态。

接受任务的图形使用一个信封图标表示，如图4-22所示。代码清单4-23列出了接受任务的XML描述。



Receive Task

图4-22 接收任务

代码清单4-23 接受任务的XML描述

4.3.11 Shell任务

Shell任务允许在流程运行过程中执行本地操作系统中的脚本、命令，是Activiti基于serviceTask扩展的一种任务（activiti:type= “shell” ）。在BPMN规范中没有对应的图形表示，Activiti添加了Shell任务的图形表示，如图4-23所示。代码清单4-24列出了Shell任务的XML描述。

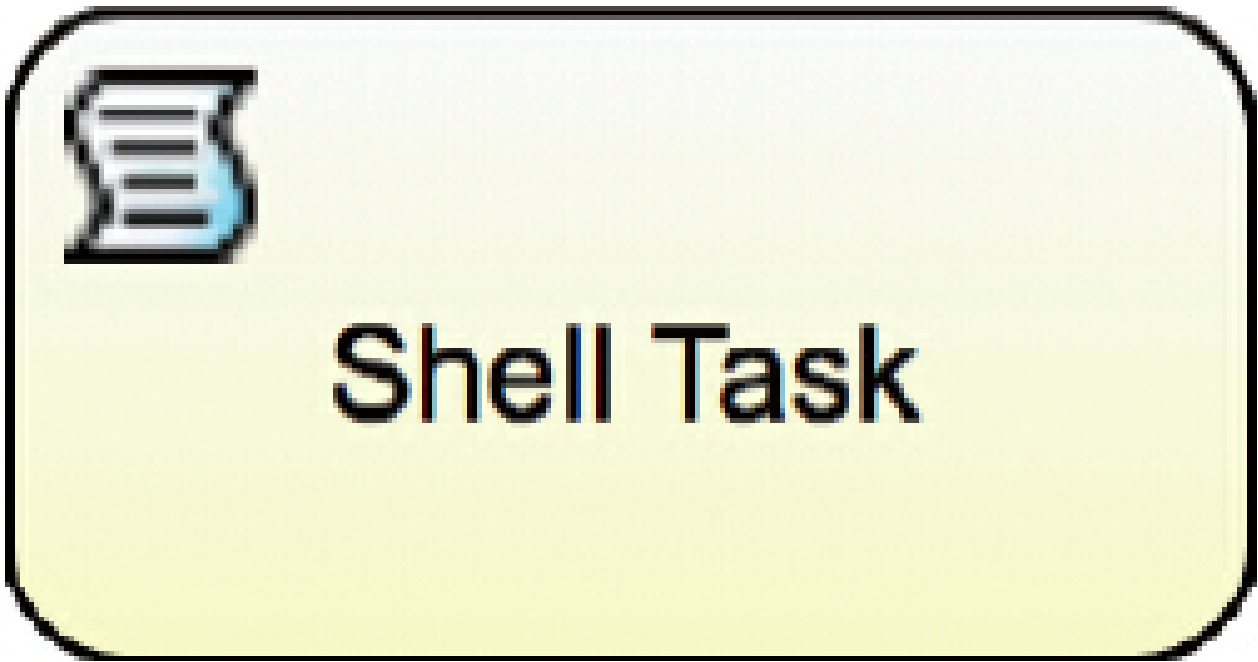


图4-23 Shell任务

代码清单4-24 Shell任务的XML描述

```
<serviceTask id="shellEcho" activiti:type="shell">
  <extensionElements>
    <activiti:field name="command" stringValue="echo" />
    <activiti:field name="arg1" stringValue="hello" />
    <activiti:field name="wait" stringValue="true" />
    <activiti:field name="outputVariable" stringValue="resultVar" />
  </extensionElements>
</serviceTask>
```

配置Shell任务的参数和邮件任务类似，使用activiti:field扩展元素定义。表4-9列举了代码清单4-24中Shell任务的属性及其含义。

表4-9 Shell任务的属性说明

属性名称	是否必选	属性类型	属性说明	默认值
command	是	String	执行的脚本、命令	
arg0-5	否	String	参数，限制6个，从0~5；在运行的时候用空格分割多个参数	
wait	否	true/false	是否等待脚本执行完成	true
redirectError	否	true/false	合并错误输出至标准输出	false
cleanEnv	否	true/false	是否继承当前的环境变量	false
outputVariable	否	String	脚本、命令执行结果保存变量	空，不记录
errorCodeVariable	否	String	脚本、命令执行失败反馈的错误编码保存变量	空，为绑定错误
directory	否	String	在哪个目录执行脚本、命令	当前项目目录

4.3.12 多实例

多实例允许业务流程中某一个任务甚至子流程可以重复执行多次，在实际应用中一个申请由多人审批是多实例的典型应用场景。多个实例可以选择顺序执行，还可以选择并行执行多实例任务或子流程。

多实例图形化描述是在原任务的基础上添加了3个垂直线（顺序执行）和3个平行线（并行执行）。图4-24和图4-25分别展示了顺序执行、并行执行的不同图形化描述。



图4-24 顺序执行的多实例

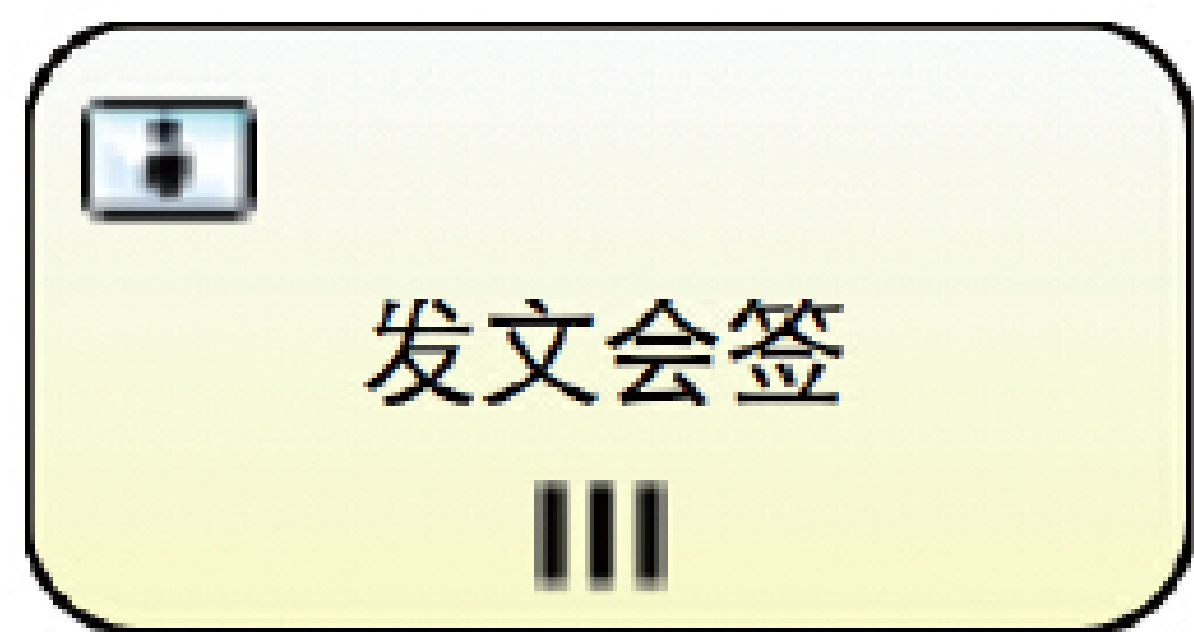


图4-25 并行执行的多实例

多实例支持的任务类型如下：

- 用户任务
- 脚本任务
- Java Service任务
- Web Service任务
- 业务规则任务
- 邮件任务
- 手动任务
- 接收任务
- 子流程（嵌入式）

· 子流程（调用活动）

 **注意** 网关和事件类型不支持多实例。

在BPMN 2.0规范中规定了多实例的几个属性变量，可以通过execution.getVariable()获取变量。

- **nrOfInstances**: 实例的总数。
- **nrOfActiveInstances**: 当前活动的（未完成的）实例数量。对于按照顺序执行的多实例，该值总是为1。
- **nrOfCompletedInstances**: 已经完成的实例数量。
- **loopCounter**: 多实例运行过程中，for-each循环中当前的索引值。

在代码清单4-25中，#1处用multiInstanceLoopCharacteristics定义了一个多实例活动，其中isSequential属性可以设置true和false两个值：值为true时按照顺序执行，值为false时并行执行；#2处定义了loopCardinality标签来指定多实例的循环次数，在并行执行的时候引擎会一次创建loopCardinality元素值数量的实例，在顺序执行时只有当一个实例完成之后才会创建下一个实例。

代码清单4-25 多实例的XML描述

```
<userTask id="audit" name="领导审批">
<multiInstanceLoopCharacteristics isSequential="true">      #1
  <loopCardinality>3</loopCardinality>      #2
</multiInstanceLoopCharacteristics>
</userTask>
```

决定启动多少个流程实例可以从两个角度设置：固定数量、任务参与人列表。表4-10列举了多实例的配置元素及属性。

表4-10 多实例的配置元素及属性

属性名称	属性说明	示例
loopCardinality	实例的数量。可以用常量设置，也可以使用表达式计算设置	<pre> <userTask id="audit" name=" 领导审批 "> <multiInstanceLoopCharacteristics isSequential="false"> <loopCardinality>3</loopCardinality> </multiInstanceLoopCharacteristics> </userTask> </pre> 此示例约定一次创建 3 个任务，也可以使用 UEL 表达式计算动态设置： <pre> <loopCardinality> \${allTaskCounter - completedTaskCounter } </loopCardinality> </pre>
loopDataInputRef	任务参与人列表。以参与人集合的数量决定创建多少个实例。和 loopCardinality 是互斥的关系	<pre> <userTask id="audit" name=" 领导审批 "> <multiInstanceLoopCharacteristics isSequential="false"> <loopDataInputRef> assigneeList </loopDataInputRef> <inputDataItem name="assignee" /> </multiInstanceLoopCharacteristics> </userTask> </pre> assigneeList 作为一个运行时变量存在，实际上是包含了一系列用户 ID 的数组
inputDataItem	需要配合 loopDataInputRef 使用，用来指定在 loop-DataInputRef 中定义的办理人集合在循环过程中单个变量的名称，例如指定 userTask 的 activiti:assignee 属性表示用户任务的办理人由变量 assignee 的值决定	<pre> <userTask activiti:assignee="\${assignee}"> <multiInstanceLoopCharacteristics isSequential="false"> <loopDataInputRef> assigneeList </loopDataInputRef> <inputDataItem name="assignee" /> </multiInstanceLoopCharacteristics> </userTask> </pre>
completionCondition	多实例循环结束条件。当满足一定条件时退出多实例循环。例如在投票表决场景中当通过率达到 70% 时结束任务，剩余未办理的任务不再执行	<pre> <userTask id="audit" name=" 领导审批 "> <multiInstanceLoopCharacteristics isSequential="false"> <completionCondition> \${ nrOfCompletedInstances/nrOfInstances >= 0.7} </completionCondition> </multiInstanceLoopCharacteristics> </userTask> </pre>
isSequential	是否按照顺序创建任务。值为 true 时表示按照顺序执行，只有在第一个任务处理完成之后才会继续创建第二个任务，以此类推；值为 false 时表示不按照顺序执行，而是一次创建多个任务（数量由其他的条件决定），这样多个任务可以同时办理，也就是并行处理	<pre> <userTask id="audit" name=" 任务分发 " activiti:assignee="henryyan"> <multiInstanceLoopCharacteristics isSequential="false"> <loopCardinality>3</loopCardinality> </multiInstanceLoopCharacteristics> </userTask> </pre> 此例在运行时一次创建 3 个任务并分配给用户 henryyan 办理。 当多个领导办理会签任务时需要设置为 false

另外 Activiti 还基于 BPMN 2.0 规范以扩展形式简化了配置，表 4-11 列举了 Activiti 扩展的多实例属性。

表 4-11 Activiti 扩展的多实例属性

属 性 名 称	属 性 说 明	示 例
activiti:collection	用来简化 BPMN 2.0 规范中的 loopDataInputRef 元素	<pre><multiInstanceLoopCharacteristics isSequential="false" activiti:collection="assigneeList"> </multiInstanceLoopCharacteristics></pre>
activiti:elementVariable	用来简化 BPMN 2.0 规范的 inputDataItem 元素	<pre><multiInstanceLoopCharacteristics isSequential="false" activiti:collection="assigneeList" activiti:elementVariable="assignee"> </multiInstanceLoopCharacteristics></pre>

[1] <http://jcp.org/en/jsr/detail?id=223>
[2] JBoss 开发的开源规则引擎, <http://www.jboss.org/drools>
[3] <http://camel.apache.org/>
[4] <http://www.mulesoft.org/>

4.4 网关

网关（Gateway）用于控制流程走向（在BPMN 2.0规范中称为“执行令牌”）。根据功能不同可以划分为以下4种网关：

- 排他网关
- 并行网关
- 包容网关
- 事件网关

4.4.1 排他网关

排他网关（Exclusive Gateway，也称做XOR Gateway）用来对流程中的决定进行建模。流程执行到该网关时，按照输出流的顺序逐个计算，当条件计算结果为true时，继续执行当前网关的输出流。

值得注意的是，在排他网关中，如果多个线路的计算结果都为true，那么只会执行第一个值为true的网关，忽略其他表达式的值为true的网关。如果多个网关计算结果没有为true的值，则引擎会抛出异常。

排他网关的图形表示形象地在菱形嵌入“X”，如图4-26所示。代码清单4-26列出了排他网关的XML描述。

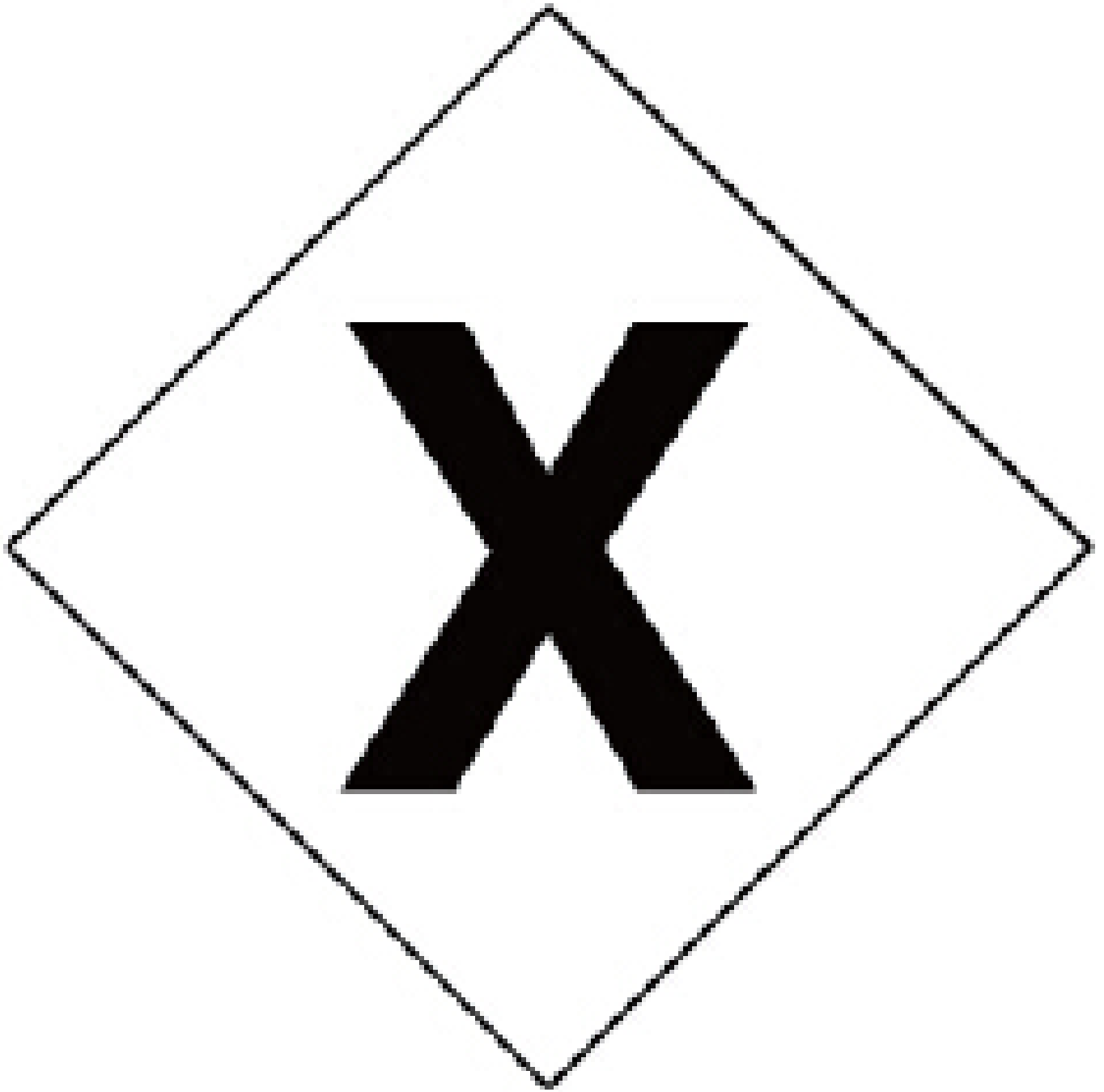


图4-26 排他网关

代码清单4-26 排他网关的XML描述

```
<exclusiveGateway id="exclusiveGateway" default="flow1"/>
```

排他网关需要和条件顺序流配合使用，一个排他网关可以连接多个条件顺序流，每个条件顺序流设置一个条件在运行时由引擎计算并根据结果是否为true决定执行与否。典型的排他网关用例如图4-27所示，对应的XML描述如代码清单4-27所示。

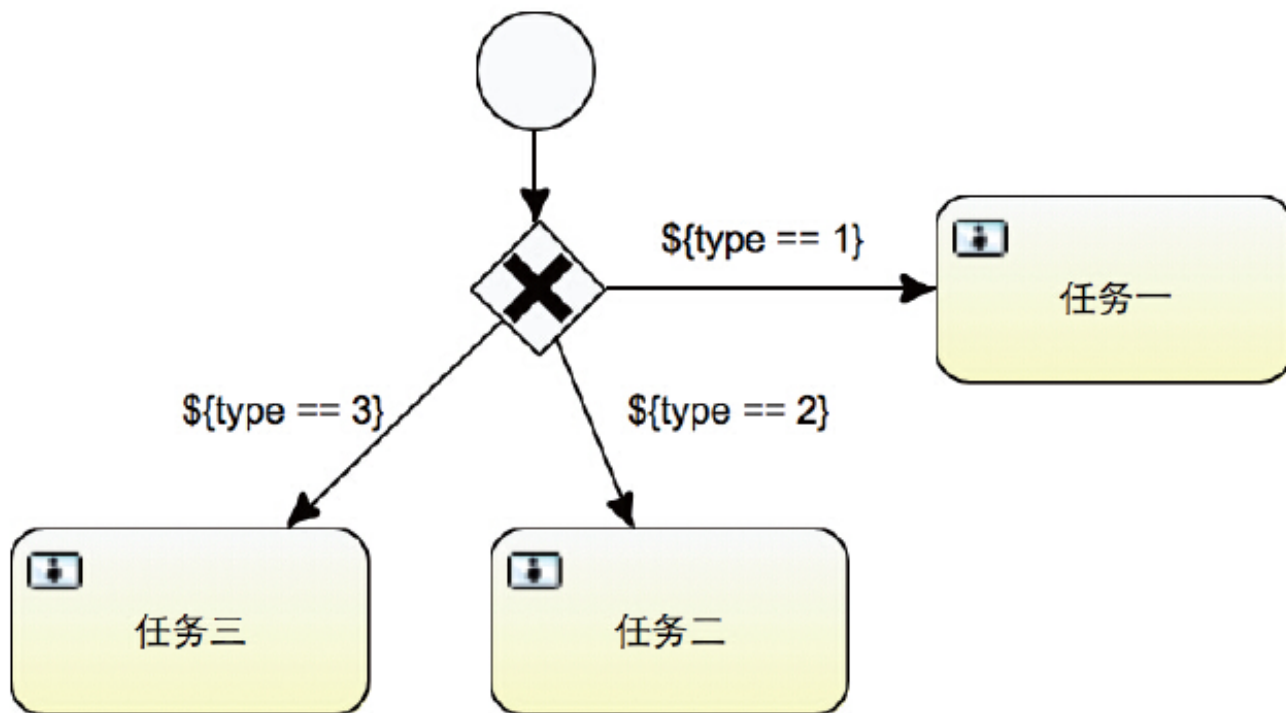


图4-27 典型的排他网关用例

代码清单4-27 图4-27对应的XML描述

```
<exclusiveGateway id="exclusivegateway1" name="Exclusive Gateway" default="flow2"> #1
</exclusiveGateway>
<sequenceFlow id="flow1" name="" sourceRef="startevent1" targetRef="exclusivegateway1">
</sequenceFlow>
<userTask id="usertask1" name="任务一"></userTask>
<sequenceFlow id="flow2" name="$ {type == 1}" sourceRef="exclusivegateway1" #2-S
targetRef="usertask1">
  <conditionExpression xsi:type="tFormalExpression">
    <![CDATA[${type == 1}]]>
  </conditionExpression>
</sequenceFlow> #2-E
<userTask id="usertask2" name="任务二"></userTask>
<sequenceFlow id="flow3" name="$ {type == 2}" sourceRef="exclusivegateway1" #3-S
targetRef="usertask2">
  <conditionExpression xsi:type="tFormalExpression">
    <![CDATA[${type == 2}]]>
  </conditionExpression>
</sequenceFlow> #3-E
<userTask id="usertask3" name="任务三"></userTask>
<sequenceFlow id="flow4" name="$ {type == 3}" sourceRef="exclusivegateway1" #4-S
targetRef="usertask3">
  <conditionExpression xsi:type="tFormalExpression">
    <![CDATA[${type == 3}]]>
  </conditionExpression>
</sequenceFlow> #4-E
```

代码清单4-27中的#1处使用exclusiveGateway定义了排他网关，并且设定id属性为exclusivegateway1，#2、#3、#4分别定义了3个条件顺序流（4.2.2节）并统一设置了sourceRef属性为exclusivegateway1，以此和排他网关进行关联。

当所有的条件都不满足时，排他网关会默认执行exclusiveGateway的default属性指定的条件顺序流（此时已经忽略了默认流的条件）。读者可以用ifhttp://www.hzcourse.com/resource/readBook?path=/openresources/teach_ebook/uncompressed/15030/OEBPS/Text/...else ifhttp://www.hzcourse.com/resource/readBook?path=/openresources/teach_ebook/uncompressed/15030/OEBPS/Text/...else方式理解排他网关。

4.4.2 并行网关

并行网关用来对并发的任务进行流程建模，它能把单条线路任务拆分（fork）成多个路径并行执行或将多条线路合并（join）。

并行网关的图形化表示是在菱形中嵌入一个加号“+”，如图4-28所示，对应的XML描述如代码清单4-28所示。

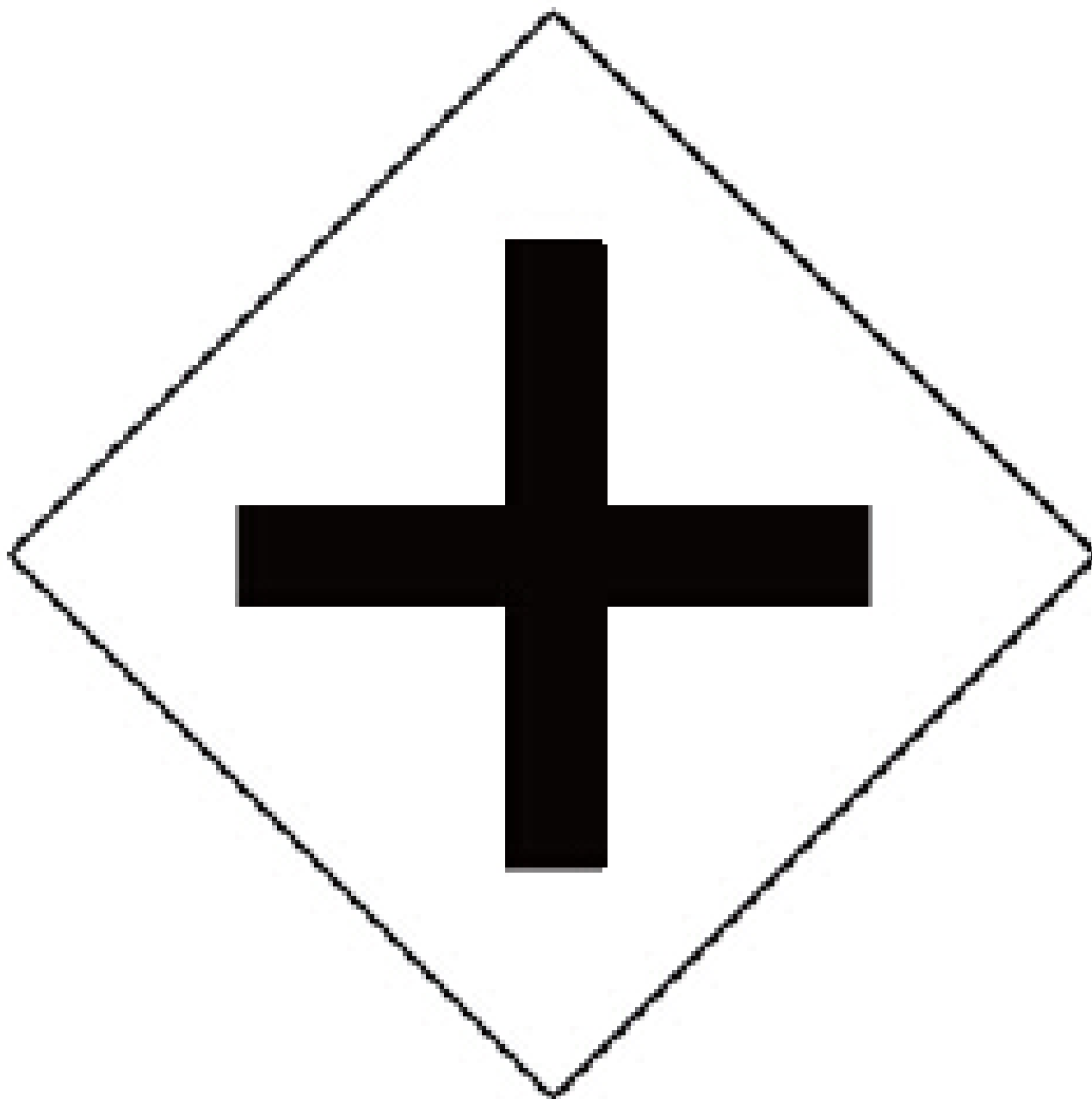


图4-28 并行网关

代码清单4-28 并行网关的XML描述

```
<parallelGateway id="fork" name="Parallel Gateway Fork">
</parallelGateway>
```

并行网关的功能取决于输入、输出顺序流。

- **拆分**：并行执行所有的输出顺序流，并且为每一条顺序流创建一个并行执行线路。
- **合并**：所有从并行网关拆分并执行完成的线路均在此等候，直到所有的线路都执行完成才继续向下执行。

 **注意** 并行网关不会计算线路上设置的条件，如果设置了，则会被直接忽略。

并行网关的图形化表示和排他网关类似，把“X”换成了“+”，加号为逻辑“与”关系运算符。典型的并行网关用例如图4-29所示。代码清单4-29列出了图4-29对应的XML描述。

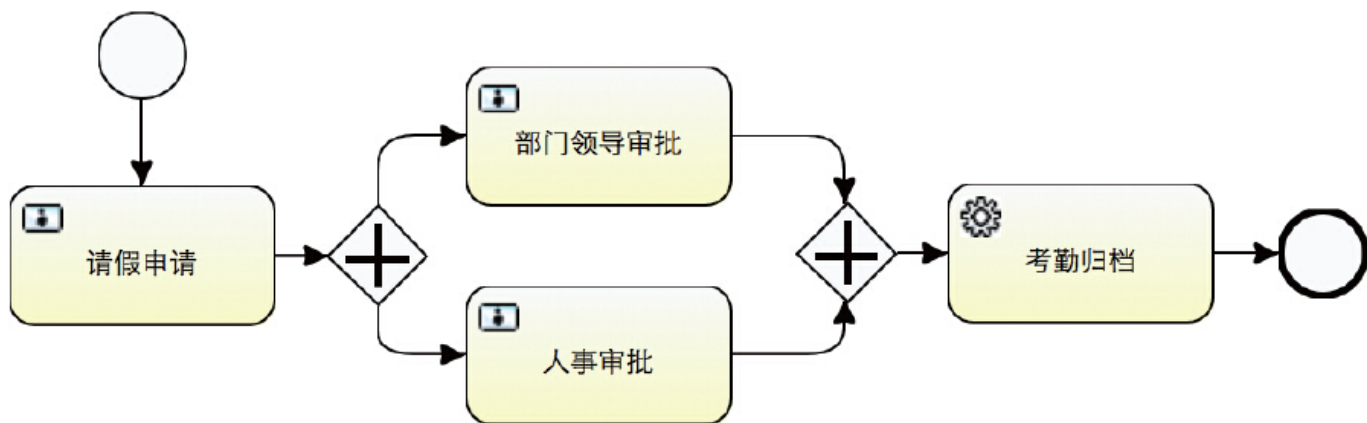


图4-29 典型的并行网关用例

代码清单4-29 图4-29对应的XML描述

```
<parallelGateway id="fork" name="Parallel Gateway Fork"></parallelGateway> #1
<userTask id="usertask1" name="部门领导审批"></userTask>
<userTask id="usertask2" name="人事审批"></userTask>
<parallelGateway id="join" name="Parallel Gateway Join"></parallelGateway> #2
<userTask id="usertask3" name="考勤归档"></userTask>
<endEvent id="endevent1" name="End"></endEvent>
<userTask id="usertask4" name="请假申请"></userTask>
<sequenceFlow id="flow2" name="" sourceRef="usertask4" targetRef="fork"/>
<sequenceFlow id="flow3" name="" sourceRef="fork" targetRef="usertask1"/> #3
<sequenceFlow id="flow4" name="" sourceRef="fork" targetRef="usertask2"/> #4
<sequenceFlow id="flow5" name="" sourceRef="usertask1" targetRef="join"/> #5
<sequenceFlow id="flow6" name="" sourceRef="usertask2" targetRef="join"/> #6
<sequenceFlow id="flow7" name="" sourceRef="join" targetRef="usertask3"/>
<sequenceFlow id="flow8" name="" sourceRef="usertask3" targetRef="endevent1"/>
<sequenceFlow id="flow9" name="" sourceRef="startevent1" targetRef="usertask4"/>
```

代码清单4-29是并行网关的基本表现形式，在#1处的`parallelGateway`描述一个并行网关，id属性为`fork`，拆分成两条线路（#3、#4处）；两个用户任务全部执行完之后在#2处合并，到此完成了`fork`和`join`的过程。

并行网关还允许在线上嵌套并行网关，也就是在`fork`拆分的线路上再添加n个`fork`线路，只要保证最后有一个`join`点合并拆分的线路即可。图4-30展示了嵌套并行网关。

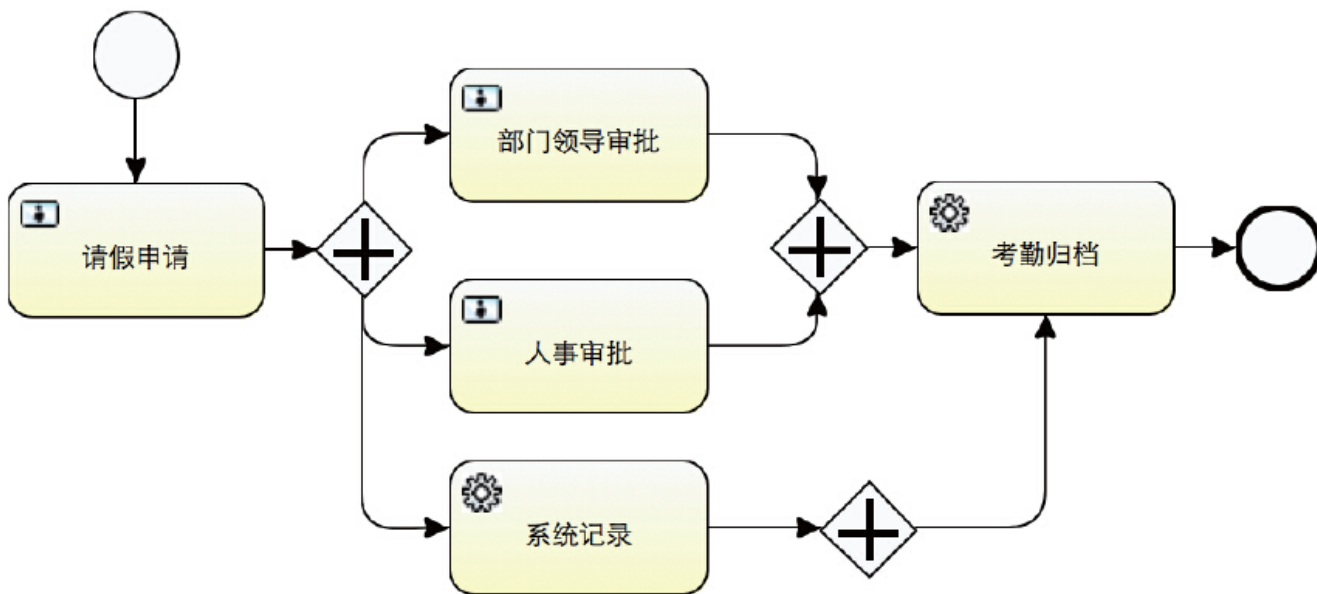


图4-30 嵌套并行网关

图4-30中的并行网关有3个，所有并行网关不需要“成对出现”，保证“有始有终”即可。

如果一个并行网关有多个输入、输出流，那么它同时具备`fork`和`join`的功能，对于这样的并行网关，首先会合并被拆分的并行网关线路，然后再拆合并网关之后的线路。图4-30中“部门领导审批”和“人事审批”节点之后的并行网关就是如此，假如“部门领导审批”节点先执行完成，此时到达“考勤归档”节点前面的并行网关，直到“人事审批”节点执行完成之后才拆分后续的“考勤归档”网关。

4.4.3 包容网关

包容网关融合了排他网关和并行网关的特性，排他网关允许在每条线路上设置条件，并行网关可以同时执行多条线路，包容网关既可以同时执行多条线路，又允许在网关上设置条件。

包容网关的图形表示形象地在菱形中嵌入“圆”，其相关的输出流均会被执行，如图4-31所示，对应的XML描述如代码清单4-30所示。

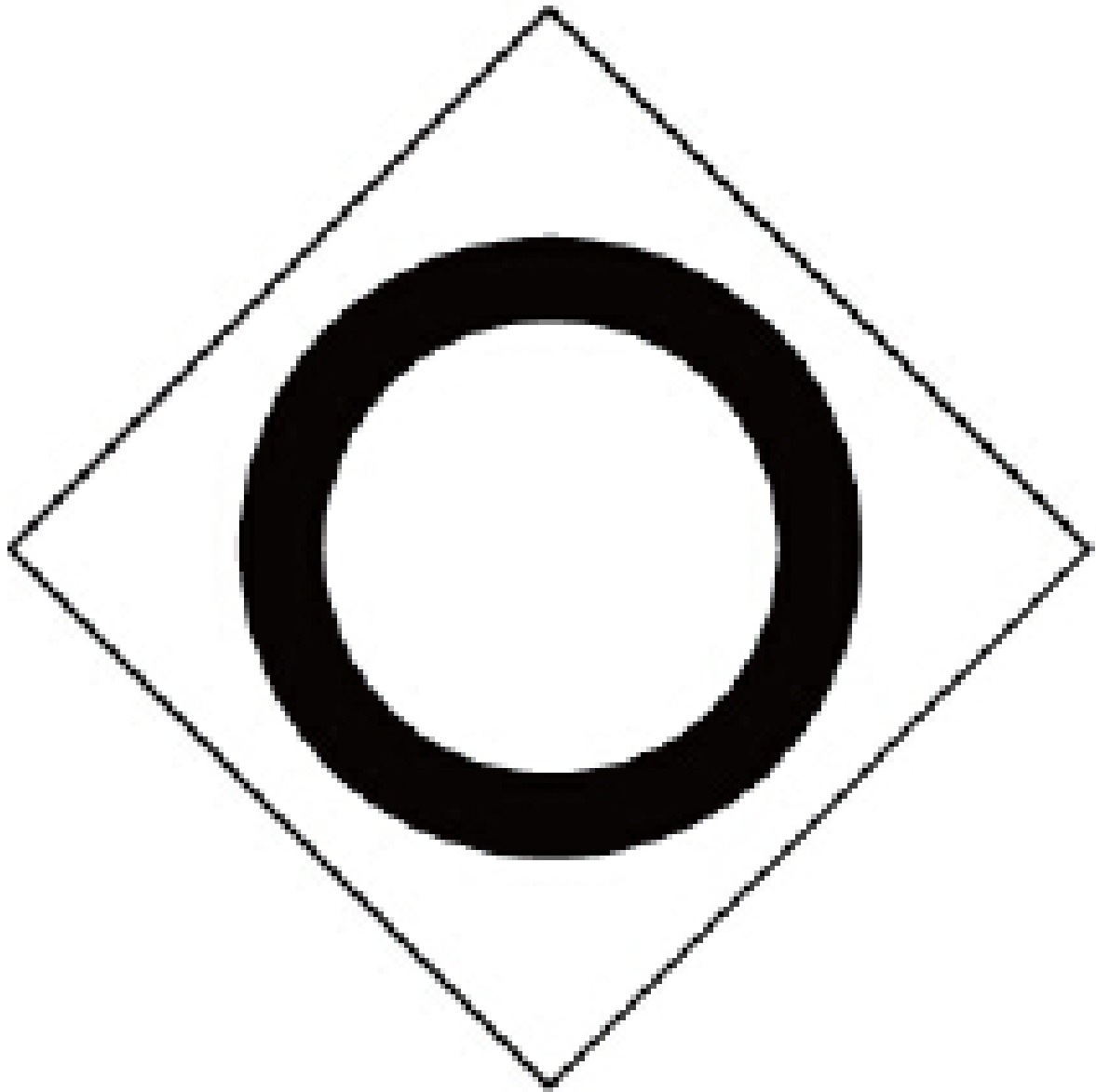


图4-31 包容网关

代码清单4-30 包容网关的XML描述

```
<inclusiveGateway id="inclusivegateway1" name="Inclusive Gateway Fork">
</inclusiveGateway>
```

和并行网关一样，包容网关的功能也取决于输入、输出顺序流。

- **拆分**：计算每条线路上的表达式，当表达式计算结果为true时，创建一个并行线路并继续执行。
- **合并**：所有从并行网关拆分并执行完成的线路均在此等候，直到所有的线路都执行完才继续向下执行。

图4-32定义了一个包容网关版本的请假流程，该流程在启动的时候根据条件顺序流判断是否需要领导或人事审批，如果两个条件都满足，则“部门领导审批”和“人事审批”任务都会被创建。

图4-32是并行网关的基本表现形式（和并行网关类似），用**inclusiveGateway**描述一个并行网关，有一个**fork**开始点拆分条件表达式计算结果为true的线路，在最后一个**join**点合并**fork**点拆分的两条线路。代码清单4-31列出了图4-32对应的XML描述。

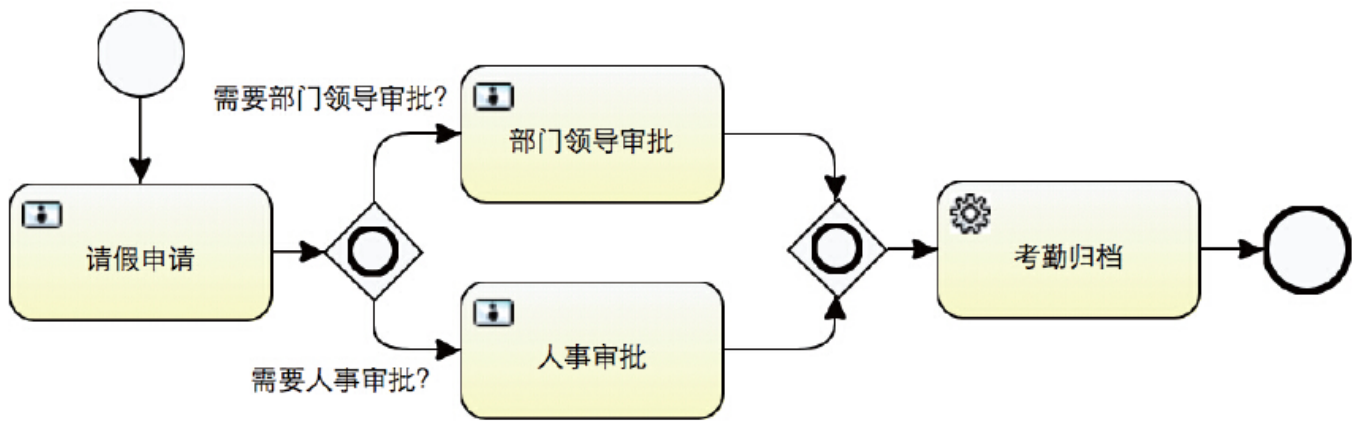


图4-32 包容网关

代码清单4-31 图4-32对应的XML描述

```
<inclusiveGateway id="igFork" name="Inclusive Gateway Fork"></inclusiveGateway> #1
<inclusiveGateway id="igJoin" name="Inclusive Gateway Join"></inclusiveGateway> #2
<userTask id="usertask1" name="部门领导审批"></userTask>
<userTask id="usertask2" name="人事审批"></userTask>
<userTask id="usertask3" name="考勤归档"></userTask>
<endEvent id="endevent1" name="End"></endEvent>
<userTask id="usertask4" name="请假申请"></userTask>
<sequenceFlow id="flow2" name="" sourceRef="usertask4" targetRef="igFork"/>
<sequenceFlow id="flow3" name="需要部门领导审批?" sourceRef="igFork" #3-S
targetRef="usertask1">
  <conditionExpression xsi:type="tFormalExpression">
    <![CDATA[${leader == true}]]>
  </conditionExpression>
</sequenceFlow> #3-E
<sequenceFlow id="flow4" name="需要人事审批?" sourceRef="igFork" #4-S
targetRef="usertask2">
  <conditionExpression xsi:type="tFormalExpression">
    <![CDATA[${hr == true}]]>
  </conditionExpression>
</sequenceFlow> #4-E
<sequenceFlow id="flow5" name="" sourceRef="usertask1" targetRef="igJoin"/> #5
<sequenceFlow id="flow6" name="" sourceRef="usertask2" targetRef="igJoin"/> #6
<sequenceFlow id="flow7" name="" sourceRef="inclusivegateway2" targetRef="usertask3"/>
<sequenceFlow id="flow8" name="" sourceRef="usertask3" targetRef="endevent1"/>
<sequenceFlow id="flow9" name="" sourceRef="startevent1" targetRef="usertask4"/>
```

代码清单4-31中的#1处定义了fork类型的包容网关；#2处定义了join类型的包容网关；#3、#4分别是id为igFork的包容网关的两个条件顺序流，在根据条件执行完两个用户任务之后汇总#5、#6两个输出流到#2处。

对比代码清单4-31和代码清单4-30可以发现，两者唯一的不同就是包容网关中的顺序流可以添加条件并在运行时可以被引擎解析；并行网关不支持条件，就算设置了条件也会被引擎忽略掉。

4.4.4 事件网关

事件网关是专门为中间捕获事件（4.6.2节）设置的，它允许设置多个输出流指向多个不同的中间捕获事件（最少两个）。在流程执行到事件网关后，流程处于“等待”状态，因为中间捕获事件需要依赖中间抛出事件（4.6.3节）触发才能更改“等待”状态为“活动”状态，当然，定时器捕获事件除外（它由时间驱动）。

事件网关的图形化表示是在菱形中嵌入了一个特殊符号（双线圆中嵌套一个多边形），如图4-33所示，对应的XML描述如代码清单4-32所示。

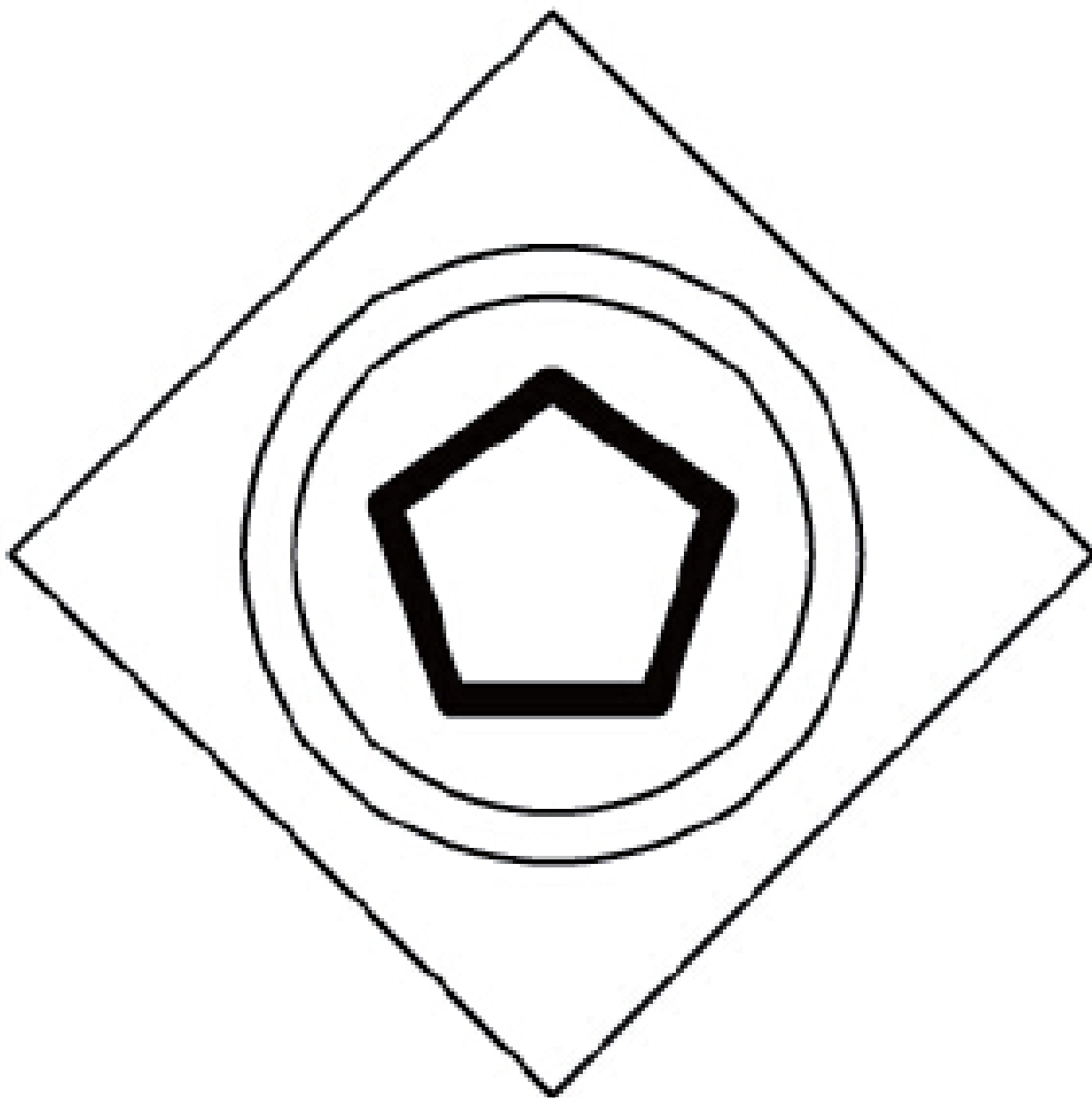


图4-33 事件网关

代码清单4-32 事件网关的XML描述

```
<eventBasedGateway id="eventgateway1" name="Event Gateway"></eventBasedGateway>
```

图4-34展示了事件网关的典型应用，其对应的XML描述如代码清单4-33所示。

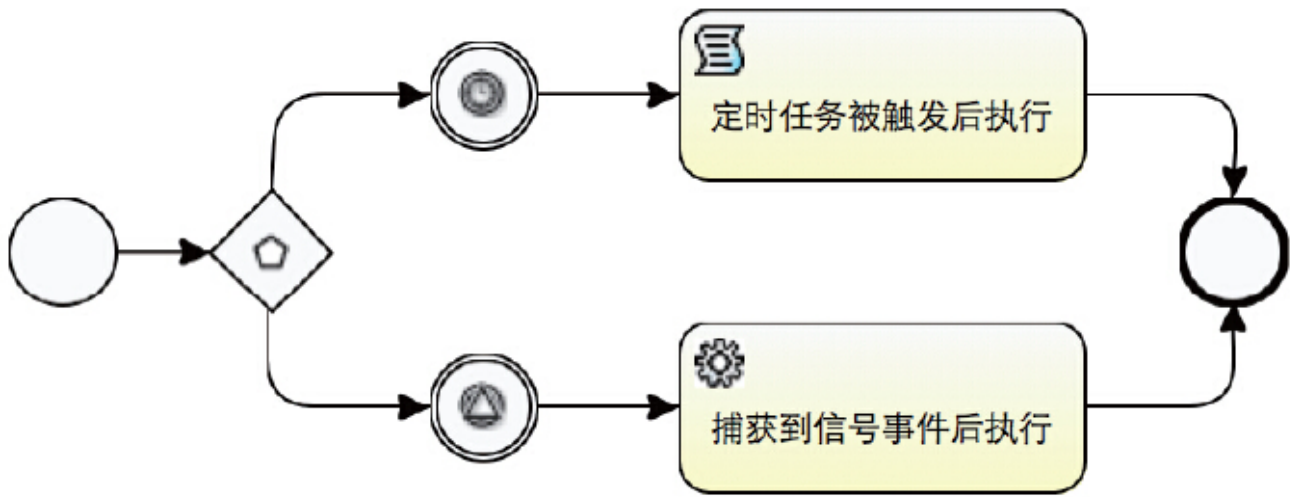


图4-34 事件网关的典型应用

代码清单4-33 图4-33对应的XML描述

```
<signal id="alertSignal" name="alert"></signal>
<process id="EventGateway" name="EventGateway">
  <startEvent id="startevent1" name="Start"></startEvent>
  <eventBasedGateway id="eventgateway1" name="Event Gateway"></eventBasedGateway> #1
  <intermediateCatchEvent id="timerintermediatecatchevent1" name="TimerCatchEvent"> #2-S
    <timerEventDefinition>
      <timeDuration>PT1S</timeDuration>
    </timerEventDefinition>
  </intermediateCatchEvent> #2-E
  <scriptTask id="scripttask1" name="定时任务执行之后" scriptFormat="groovy">
    <script><![CDATA[out:println "after time event";]]></script>
  </scriptTask>
  <intermediateCatchEvent id="signalintermediatecatchevent1" name="SignalCatchEvent"> #3-S
    <signalEventDefinition signalRef="alertSignal"></signalEventDefinition>
  </intermediateCatchEvent> #3-E
  <scriptTask id="scripttask2" name="信号捕获事件之后" scriptFormat="groovy">
    <script><![CDATA[out:println "after signal event";]]></script>
  </scriptTask>
  <endEvent id="endevent2" name="End"></endEvent>
  <sequenceFlow id="flow1" name="" sourceRef="startevent1"
    targetRef="eventgateway1"></sequenceFlow>
  <sequenceFlow id="flow2" name="" sourceRef="eventgateway1"
    targetRef="timerintermediatecatchevent1"></sequenceFlow>
  <sequenceFlow id="flow3" name="" sourceRef="timerintermediatecatchevent1" targetRef="scripttask1"/>
  <sequenceFlow id="flow4" name="" sourceRef="scripttask1" targetRef="endevent2"/>
  <sequenceFlow id="flow5" name="" sourceRef="eventgateway1"
    targetRef="signalintermediatecatchevent1"></sequenceFlow>
  <sequenceFlow id="flow6" name=""
    sourceRef="signalintermediatecatchevent1"
    targetRef="scripttask2"></sequenceFlow>
  <sequenceFlow id="flow7" name="" sourceRef="scripttask2"
    targetRef="endevent2"></sequenceFlow>
</process>
```

代码清单4-33首先在#1处定义了一个事件网关（`eventBasedGateway`），分别指向两个输出流（对应#2、#3），分别为定时器中间捕获事件（4.6.2节）和信号中间捕获事件（4.6.2节），在流程启动之后两个输出流均处于“等待”状态。定时器中间事件通过`timeDuration`指定定时器在1秒钟之后执行；信号中间事件设置了`signalRef="alertSignal"`，等待其他流程中的信号抛出事件抛出`alertSignal`的信号。中间捕获事件在满足了条件后会被自动触发继续执行输出流的活动。

关于事件网关，需要注意几点：

- 事件网关的输出流数量必须大于2个。
- 事件网关的输出流类型只能是中间捕获事件，Activiti不支持接受任务（4.3.10节）后面的事件网关。
- 中间捕获事件的输出流只能有一个。

4.5 子流程与调用活动

在实际的企业应用中，业务流程往往比较复杂，仔细分析之后一般可以将其划分为多个不同的阶段，可以把这些阶段规划为一个子流程作为主流程的一部分，BPMN 2.0的子流程规范正是满足此需求的选择之一。

在企业中还有一些通用的业务流程，例如，付款流程作为公司业务运作的核心流程之一，在业务设计及架构设计上会保持通用，或者在业务架构中作为一个通用的模块，不同的业务根据财务流程的规范传入指定的参数就可以使用付款流程。调用活动的特点和子流程类似，但是子流程嵌入在主流程中，要保持通用需要把付款流程作为活动由主流程“调用”，如此调用活动既包含了子流程的特性又保持通用。

4.5.1 子流程

子流程可以包含BPMN 2.0规范中的大部分模型，把一系列需要处理的任务归结到一起，作为一个大流程中的一部分。因为子流程嵌入在主流程中，所以也把子流程称之为“嵌入式子流程”。

子流程需要把一系列模型包含在一个实线矩形中，形成如图4-35所示的子流程，对应的XML描述如代码清单4-34所示。

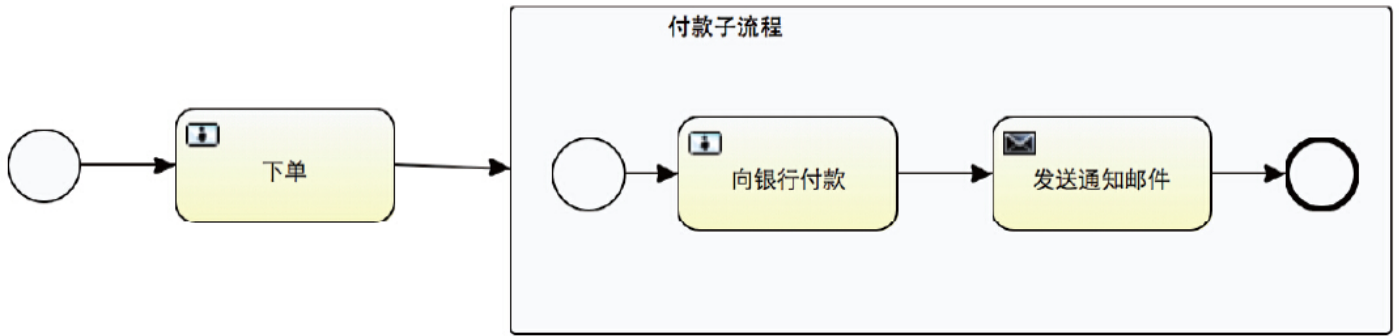


图4-35 子流程的应用场景

代码清单4-34 子流程的XML描述

```
<subProcess id="subprocess1" name="付款子流程">
...
</subProcess>
```

对于子流程BPMN 2.0对其做了一些限制。

- 只能且仅能包含一个空启动事件。
- 至少要有有一个结束事件（每个流程都要“有始有终”）。
- 在子流程中顺序流不能直接设置输出流到子流程之外的活动上，如果需要的，可以通过边界事件代替。

注意 BPMN 2.0规范允许子流程不包含启动、结束事件，但是目前Activiti还不支持，所以本书中所有提到的子流程均包含启动、结束事件。

代码清单4-35 图4-35对应的XML描述

```
<process id="subprocess" name="subprocess">
<startEvent id="startevent1" name="Start"></startEvent>
<userTask id="usertask1" name="下单"></userTask>
<subProcess id="subprocess1" name="付款子流程" #1
<startEvent id="startevent2" name="Start"></startEvent>
<userTask id="usertask2" name="银行付款"></userTask>
<endEvent id="endevent1" name="End"></endEvent>
<serviceTask id="mailtask1" name="发送邮件通知" activiti:type="mail">
...
</serviceTask>
<sequenceFlow id="flow3" name="" sourceRef="startevent2" targetRef="usertask2"/>
<sequenceFlow id="flow9" name="" sourceRef="mailtask1" targetRef="endevent1"/>
<sequenceFlow id="flow10" name="发送付款结果" sourceRef="usertask2" targetRef="mailtask1"/>
</subProcess> #2
<sequenceFlow id="flow1" name="" sourceRef="startevent1" targetRef="usertask1"/>
<sequenceFlow id="flow2" name="" sourceRef="usertask1" targetRef="subprocess1"/>
<endEvent id="endevent2" name="End"></endEvent>
<sequenceFlow id="flow11" name="" sourceRef="subprocess1" targetRef="endevent2"/>
</process>
```

代码清单4-35中的#1处使用subProcess包裹了启动、结束及用户任务作为一个子流程。读者可以把子流程作为一个独立的流程来理解，只不过有一些限制和表现形式不同而已。在#2处结束子流程。

在实际运行中流程引擎会自动为主流程和子流程建立关联关系，子流程可以通过API获取主流程的一些信息及变量。

注意 子流程不能直接越界指定输出流，也就是说子流程中的输出流只能指向在实践边框内的活动，如果因为异常需要终止子流程的执行可以通过4.6.1节介绍的边界事件处理。

子流程同时也支持多实例特性，例如部门领导为几个业务员分配任务，而每项任务都是一个子流程，部门领导下发任务之后每个业务员都会拥有一个独立的子流程实例，等所有的业务员都处理完成之后再汇总给部门领导，这在第9章中将会详细讲解。

4.5.2 调用活动

调用活动（Call Activity）解决的问题是流程的通用性。和子流程的作用一致，但是表现方式不同，使用一个调用活动取代嵌入子流程方式的活动即可，通过创建一个调用活动模型并指定外部流程的ID方式作为主流程的一个子活动。图4-36展示了调用活动的图形化表示，对应的XML描述如代码清单4-36所示。

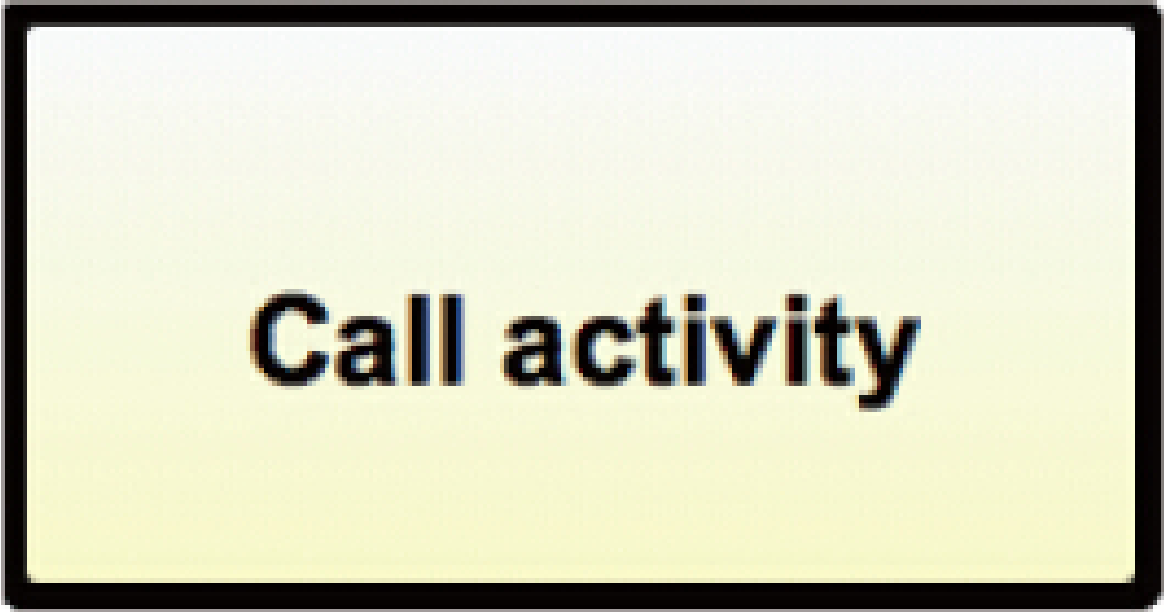


图4-36 调用活动

代码清单4-36 调用活动的XML描述

```
<callActivity id="callactivity1" name="Call Activity" calledElement="payment"></callActivity>
```

表4-12列出了调用活动和子流程的区别。

表4-12 子流程与调用活动的区别

区 别	子 流 程	调 用 活 动
表现形式	直接嵌入在主流程中，使用 subProcess 定义子流程	作为一个普通的模型，定义外部流程的 ID
模型约束	启动事件只能使用空启动事件	无任何限制，被调用的外部流程本身就是一个完整的流程
流程变量	子流程共享主流程的所有变量	需要指定输入、输出变量

图4-37展示了调用活动的用例，其对应的XML描述如代码清单4-37所示。

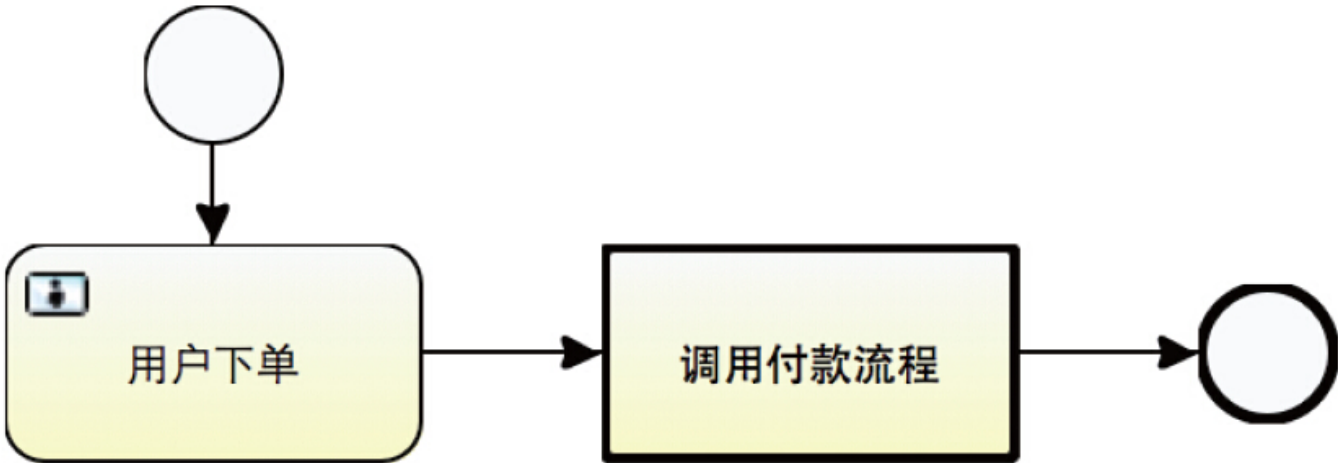


图4-37 调用活动的用例

代码清单4-37 图4-37对应的XML描述

```
<callActivity id="callactivity1" name="调用付款流程" calledElement="payment"> #1
```

```
<extensionElements>
  <activiti:in source="amount" target="amount"></activiti:in>          #2
  <activiti:out source="paid" target="paid"></activiti:out>          #3
  <activiti:out sourceExpression="{payTime}" target="payTime"></activiti:out>  #4
</extensionElements>
</callActivity>
```

代码清单4-37中调用了外部的付款流程，名称为payment；指定了输入、输出参数。表4-13列出了调用活动的属性及Activiti的扩展元素。

表4-13 调用活动的属性及Activiti的扩展元素

属性名称	属性说明	示例
calledElement	流程的 ID，对应的流程应独立存在。见代码清单 4-37 中 #1 处	<callActivity id="callactivity1" name=" 调用付款流程 " calledElement="payment">
activiti:in	调用外部流程时传入的变量，被调用活动需要获取主流程的信息（必须显示定义，否则不能获取）。见代码清单 4-37 中 #2 处	<activiti:in source="amount" target="amount"></activiti:in> ❑ source ：主流程中的变量名称，也可以在使用表单时计算 ❑ target ：在将变量传递给调用活动时使用的名称，一般和 source 同名避免错误的出现
activiti:out	调用活动执行完成后的结果。见代码清单 4-37 中 #3、#4 处	<activiti:in source="amount" target="amount"></activiti:in> ❑ source ：主流程中的变量名称，也可以在使用表单时计算 ❑ target ：在将变量传递给调用活动时使用的名称，一般和 source 同名避免错误的出现

4.5.3 事件子流程

事件子流程和子流程类似，把一系列的活动归结到一起处理，不同的是事件子流程不能直接启动，而要“被动”地由其他的事件触发启动；在图形化表示上有所不同，子流程使用的是实线边框，而事件子流程使用的是虚线矩形，如图4-38所示，对应的XML描述如代码清单4-38所示。

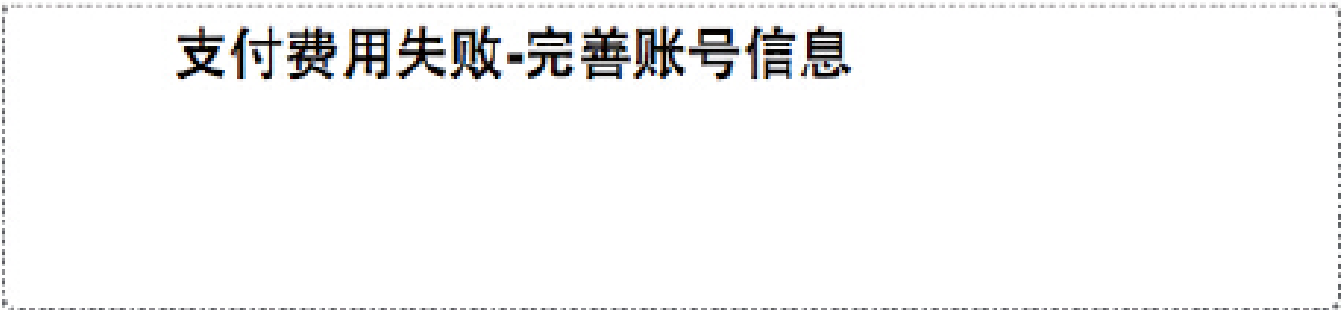


图4-38 事件子流程

代码清单4-38 事件子流程的XML描述

```
<subProcess id="eventsubprocess1" name="完善账号信息" triggeredByEvent="true">
...
</subProcess>
```

事件子流程在XML描述上和子流程不同的是添加了一个**triggeredByEvent**属性表示此子流程只能由事件触发后被动启动。

事件子流程在企业中使用频率很高，它可以由异常事件、信号事件、消息事件、定时器事件、补偿事件等触发，从而启动一个子流程。

事件子流程和子流程不同，子流程作为主流程中输出流的一个输出活动，而事件子流程是独立在主流程中的，图4-39是用付款业务模拟的一个事件子流程的应用场景。

图4-38的用例在支付失败时用排他网关把输出流转向异常结束事件，主流程中的“支付费用失败——完善账号信息”作为一个异常处理子流程存在，在4.1.1和4.1.2中分别讲解了异常启动事件和异常结束事件，此例也恰好演示了两者的联合应用。

 **注意** 异常结束事件和异常启动事件的errorRef要保持一直。如果主流程中有多个异常可以添加多个事件子流程，那么要分别处理。

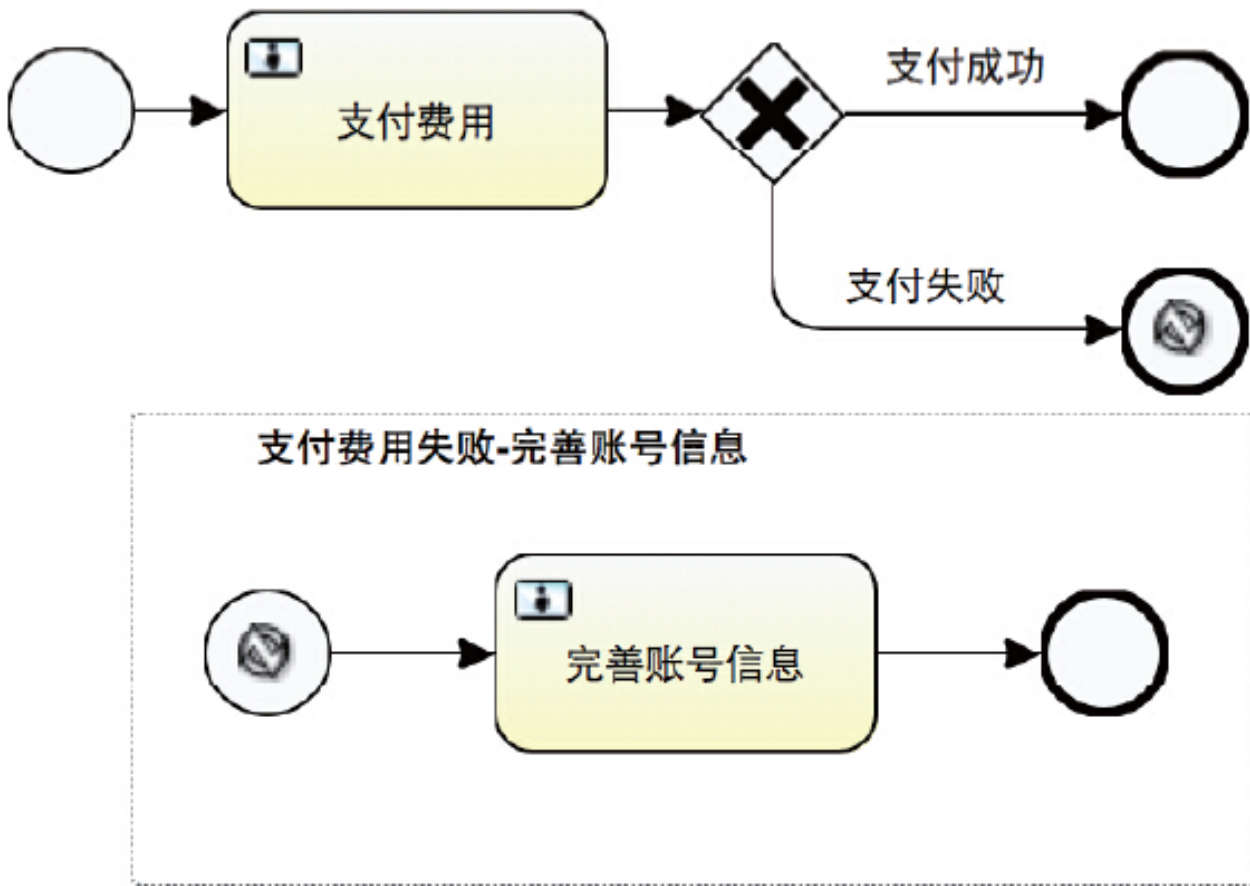


图4-39 用付款业务模拟的一个事件子流程的应用场景

代码清单4-39 图4-39对应的XML描述

```

<process id="process1" name="process1">
  <startEvent id="startevent1" name="Start"/></startEvent>
  ...
  <endEvent id="endevent3" name="ErrorEnd">
    <errorEventDefinition errorRef="A002"/></errorEventDefinition> #1
  </endEvent>
</process>
<subProcess id="eventsubprocess1"
  name="支付费用失败--完善账号信息" triggeredByEvent="true"> #2
  <startEvent id="errorstartevent1" name="Error start">
    <errorEventDefinition errorRef="A002"/></errorEventDefinition> #3
  </startEvent>
  <userTask id="usertask2" name="完善账号信息"/></userTask>
  <sequenceFlow id="flow6" name="" sourceRef="errorstartevent1" targetRef="usertask2"/>
  <endEvent id="endevent4" name="End"/></endEvent>
  <sequenceFlow id="flow7" name="" sourceRef="usertask2" targetRef="endevent4"/>
</subProcess>

```

代码清单4-39中的#2处通过在`subProcess`标签添加属性`triggeredByEvent=true`定义了一个事件子流程，#1处使用一个异常结束事件（4.1.2节）抛出异常，#3处用来捕获一个异常事件，因为`errorRef`的值为“A002”，所以当主流程“process1”执行到ID为“endevent3”的异常结束事件时引擎会启动子流程“eventsubprocess1”。注意代码清单4-39中加粗的部分，错误编号要保持一致。

付款（事务）子流程

图4-40 事务子流程

4.5.4 事务子流程

事务子流程也称为“事务块”，用来处理一组必须在同一个事务中完成的活动，这些活动要么一起成功，要么一起失败。事务子流程中的活动具有ACID^[1]（数据库事务的4个特性）特性，如果其中一个活动失败或取消，则整个事务子流程的所有活动全部回滚。

事务子流程的图形化表示采用双线矩形，如图4-41所示，对应的XML描述如代码清单4-40所示。

代码清单4-40 事务子流程的XML描述

```
<transaction id="myTransaction" >
  http://www.hzcourse.com/resource/readBook?path=/openresources/teach_ebook/uncompressed/15030/OEBPS/Text/...
</transaction>
```

图4-41展示了事务子流程的一个应用场景——银行汇款。

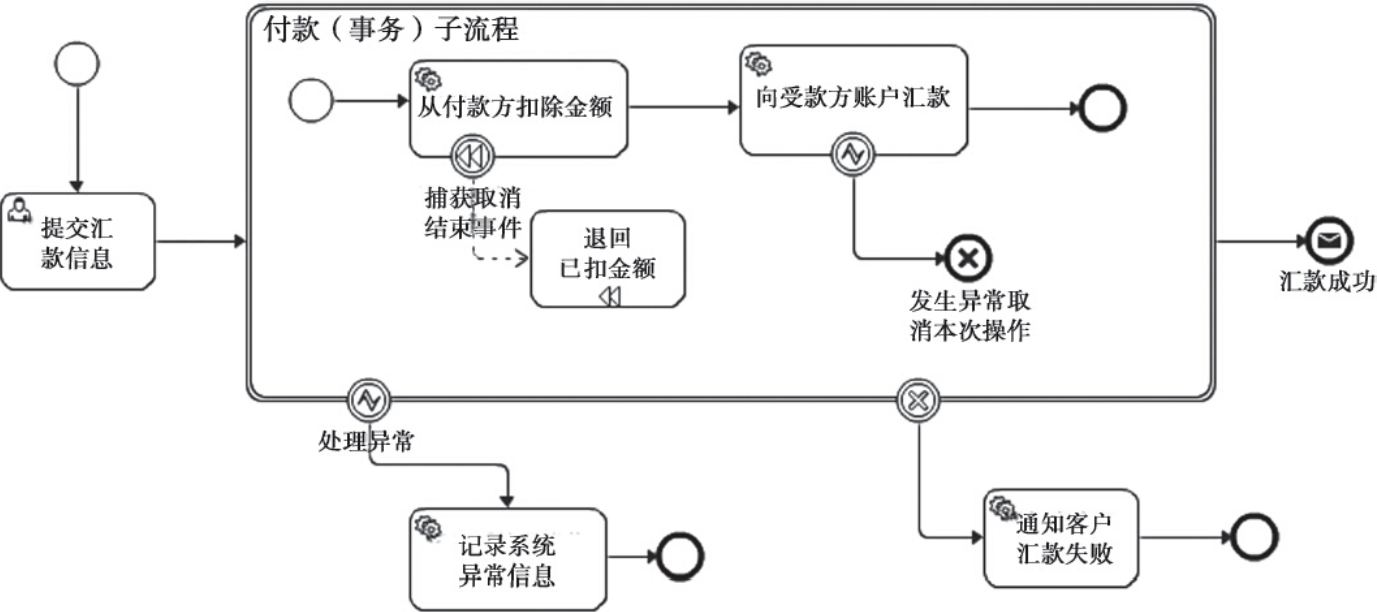


图4-41 事务子流程应用场景——银行汇款

图4-41模拟了一个简单的银行汇款流程，在主流程中嵌入了一个事务子流程把汇款业务放在一个事务中处理。根据事务执行结果可以把执行结果划分为以下几种。

- **事务成功**：事务子流程中的两个任务全部成功，正常结束。
- **事务取消**：在事务执行过程中出现异常，任务“向收款方账户汇款”抛出了异常，接着被附加在此任务边界上的异常边界事件捕获，然后执行后续输出流的取消结束事件，最后被附加在事务子流程中的取消边界事件捕获执行“通知客户汇款失败”任务；与此同时会触发子流程中的补偿边界事件执行事务取消的补偿逻辑（任务“退回已扣金额”）。在图4-41中，在“向收款方账户汇款”节点的执行过程出现异常被捕获之后执行取消结束事件（4.1.2节），随即被附加在事务子流程的取消边界事件（4.6.1节）捕获。
- **事务失败**：事务执行时遇到不可处理的异常，既不能补偿又不能回滚事务。在图4-41中捕获到异常之后会执行“记录系统异常信息”活动。

[1] <http://zh.wikipedia.org/zh-hk/ACID>

4.6 边界与中间事件

除了4.1节中提到的启动和结束事件之外，中间事件提供的特殊功能可以用来处理流程执行过程中抛出、捕获的事件，具体包括边界事件、中间捕获事件、中间抛出事件。每种中间事件的图形表示都有一个共同特点：以双线圆形表示，如图4-42所示。

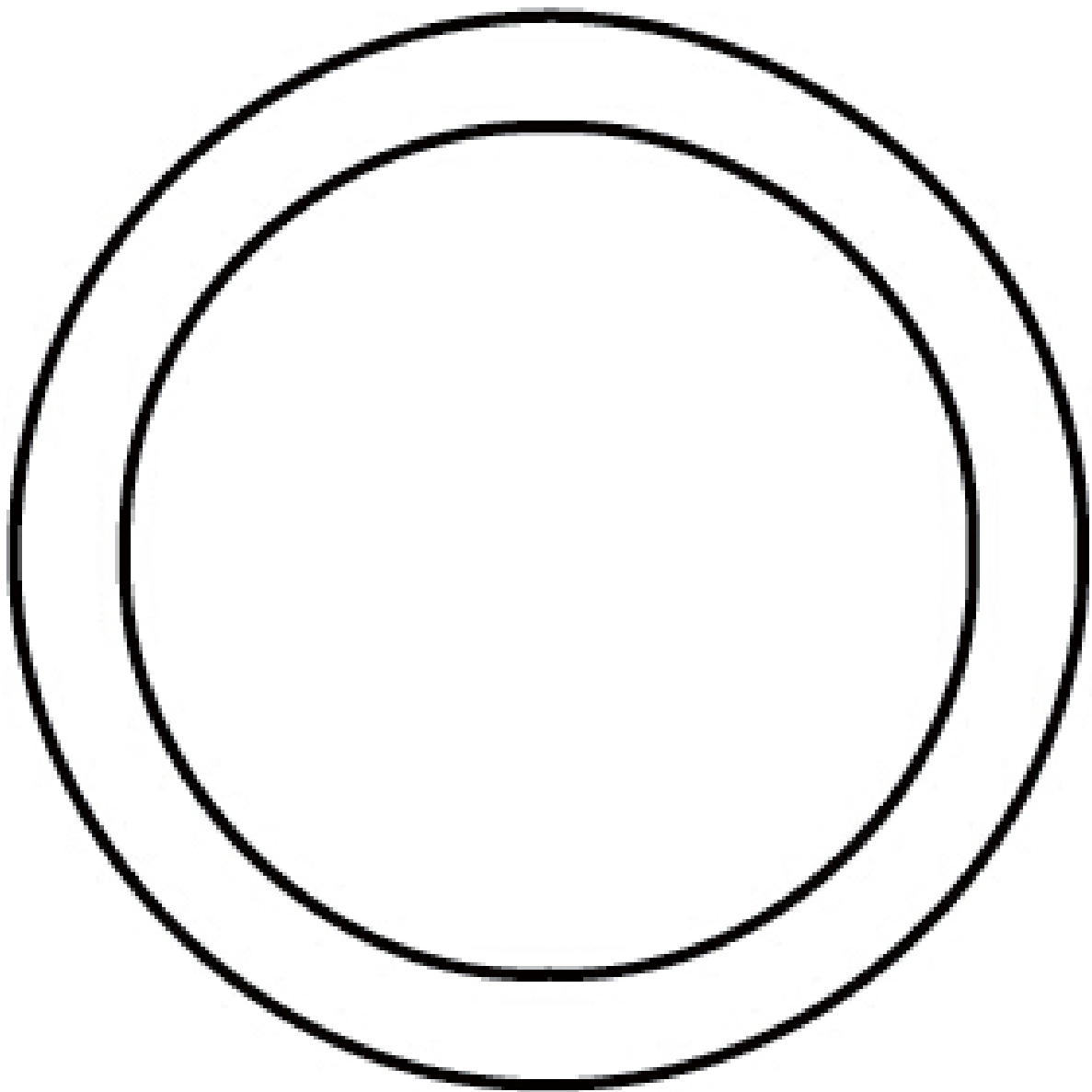


图4-42 边界与中间事件的基本图形

4.6.1 边界事件

边界事件是绑定在活动上的“捕获型”事件，会一直监听所有处于运行中活动的某种事件的触发，在捕获到事件之后中断活动，然后从边界事件类型的输出流继续执行。

值得注意的是，一旦触发边界事件，当前的活动就会被中断，然后按照边界事件之后的输出流执行。

边界事件和所关联的活动有一个特殊的关系“附加”，而且一个活动只能绑定一个边界事件；每个边界事件类型都是通过属性`attachedToRef`指定“附加”到抛出边界事件的活动上。

代码清单4-41列出了边界事件的基本XML描述。

代码清单4-41 边界事件的基本XML描述

```
<boundaryEvent id="boundary" attachedToRef="someActivity" cancelActivity="true|false">
  <xxxxEventDefinition />
</boundaryEvent>
```

所有的边界事件子类型均需要包含在`boundaryEvent`标签中。`cancelActivity`属性可以取`true`、`false`两个值，用来指定在捕获到边界事件之后是否取消执行输出流指定的活动，不过此属性仅适用部分边界事件，在讲解过程中会特别说明。

1. 定时器边界事件

定时器边界事件和定时启动事件类似，只不过两者的应用场合不同，定时启动事件用于在一个指定的时间启动一个新的流程，而定时器边界事件需要附属在一个非自动任务（用户任务等）、调用活动、子流程上，在上游任务执行完成之后开始倒计时预设的时间，到达预设时间之后触发定时器边界事件的输出流（输出流可以是并行）。

定时器边界事件的图形化表示和定时器启动事件类似，在边界事件的基础上添加一个事件图标，如图4-43所示。



图4-43 定时器边界事件

图4-44展示了定时器边界事件在请假流程中的应用，对应的XML描述如代码清单4-42所示。

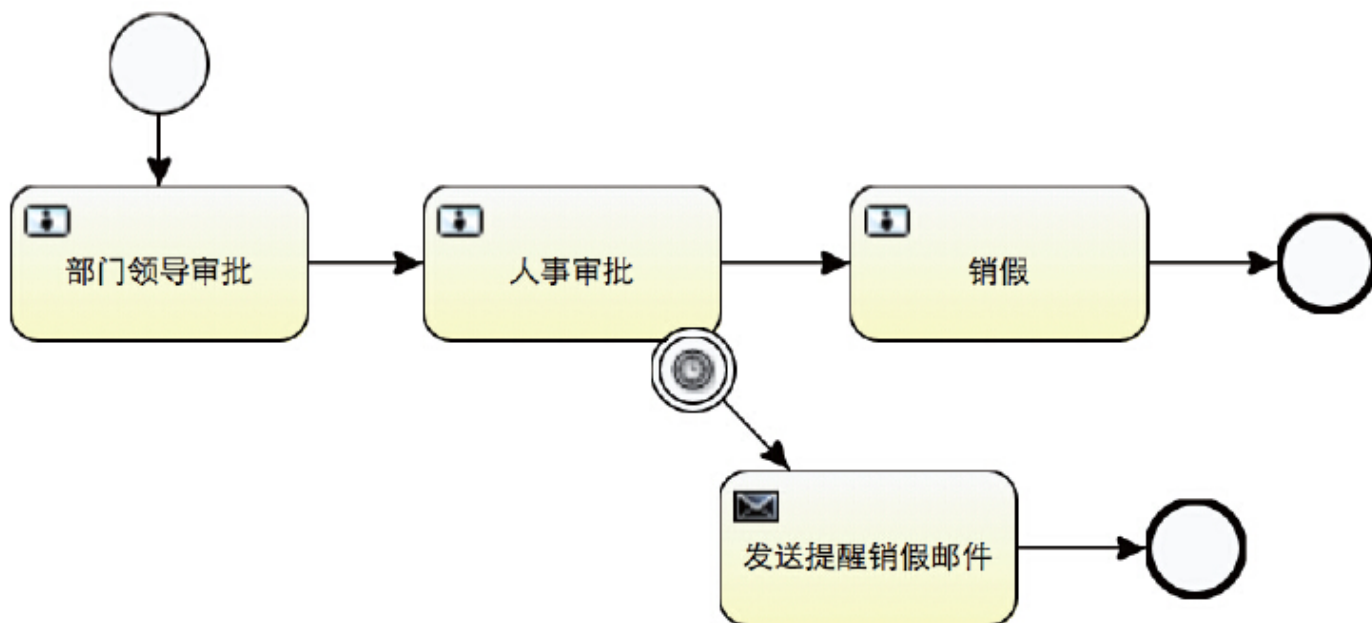


图4-44 定时器边界事件在请假流程中的应用

代码清单4-42 图4-44中定时器边界事件的XML描述

```
<boundaryEvent id="boundarytimer1" cancelActivity="false" attachedToRef="hrAudit">
  <timerEventDefinition>
    <timeDuration>P3D</timeDuration>
  </timerEventDefinition>
</boundaryEvent>
```

对于timeDuration的设置可参考4.1.1节对定时启动事件的介绍。

2.异常边界事件

异常边界事件用来捕获嵌入子流程或调用活动（4.5节）抛出的异常。异常在抛出之后被主流程的异常边界事件捕获，同时嵌入子流程或调用活动中的活动也被中断执行。

异常边界事件的图形化表示是在中间事件的基础上嵌入了一个“雷电”符号，如图4-45所示。



图4-45 异常边界事件

图4-46展示了一个网上商场的模拟流程，包含了异常边界事件处理。

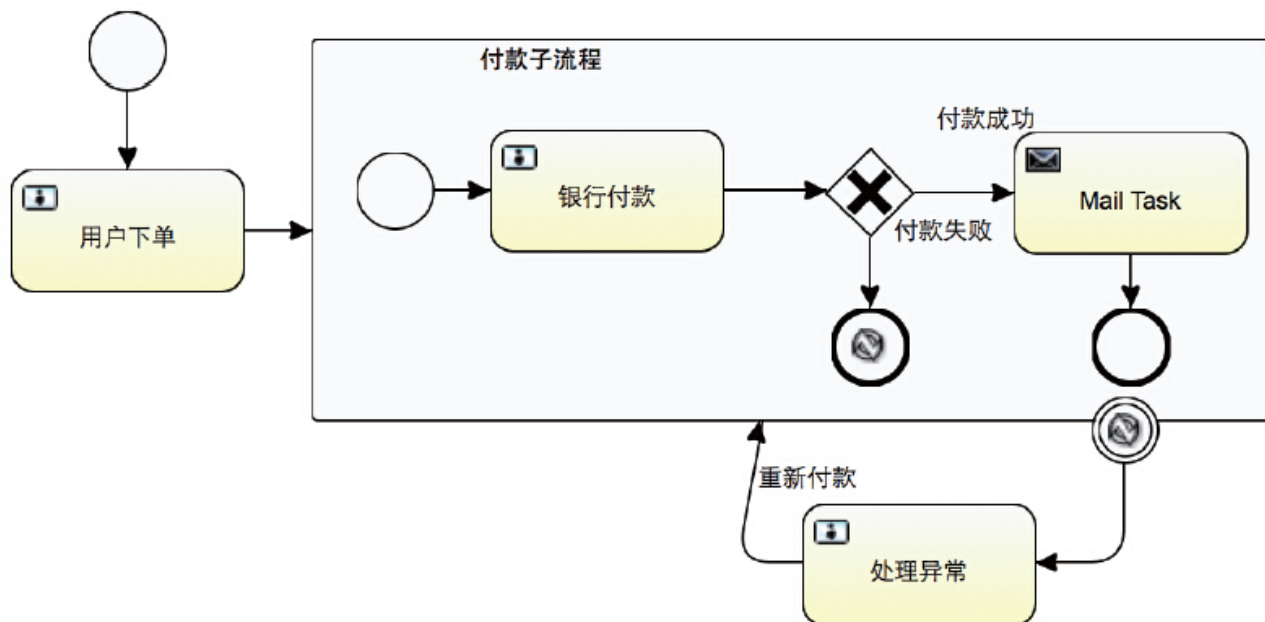


图4-46 网上商场的模拟流程

图4-46对应的XML描述如代码清单4-43所示。

代码清单4-43 图4-46的子流以及边界事件的XML描述

```
<subProcess id="subprocess1" name="付款子流程">
  http://www.hzcourse.com/resource/readBook?path=/openresources/teach_ebook/uncompressed/15030/OEBPS/Text/...
  <endEvent id="endevent2" name="ErrorEnd"> #1-S
    <errorEventDefinition errorRef="A001">
      </errorEventDefinition>
    </endEvent>
    #1-E
  </subProcess>
  <boundaryEvent id="boundaryerror1" cancelActivity="false" #2-S
    attachedToRef="subprocess1"> #2-1
      <errorEventDefinition errorRef="A001" /> #2-E
    </boundaryEvent>
```

从代码清单4-43的#2-1处可以看出，attachedToRef指定到了子流程subprocess1，在子流程中“银行付款”节点出现异常之后退出付款子流程，紧接着被“附加”在子流程的异常边界事件捕获并执行唯一的输出流“处理异常”任务，#1处用于抛出一个异常结束事件，#2用于捕获一个异常事件。

付款子流程的异常结束事件和4.1.1中的“异常启动事件”相同，也是通过errorRef指定错误类型，在本例中，在boundaryEvent的errorEventDefinition元素中定义了相同的错误引用编码“A001”。

3.信号边界事件

信号边界事件可以捕获流程执行过程中抛出的信号，可以“附加”在各种活动和子流程上。

信号边界事件不仅可以捕获本流程的信号，还可以捕获到其他流程的信号事件，而且如果在一个活动或子流程上定义了多个信号边界事件并监听同一个信号，则会同时触发，因为对应的信号抛出事件（4.6.3节）是全局的。



图4-47 信号边界事件

信号边界事件的图形化表示在边界事件的基础上嵌入了三角形，如图4-48所示，对应的XML描述如代码清单4-44所示。

代码清单4-44 信号边界事件的XML描述

```
<boundaryEvent id="boundary" attachedToRef="task" cancelActivity="true">  
  <signalEventDefinition signalRef="alertSignal"/>  
</boundaryEvent>
```

4.取消边界事件

取消边界事件是专门针对事务子流程所设立的，用来捕获子流程中抛出的取消事件，读者可以结合4.5.4节来理解。

取消边界事件的图形表示在中间事件的基础上嵌入空心的“X”，如图4-48所示，对应的XML描述如代码清单4-45所示。



图4-48 取消边界事件

值得注意的是，取消边界事件只能和事务子流程结合使用，不能附加在其他的活动上。

代码清单4-45 取消边界事件的XML描述

```
<boundaryEvent id="boundary" attachedToRef="transaction" >
  <cancelEventDefinition />
</boundaryEvent>
```

在结合事务子流程使用时需要注意几点：

- 一个事务子流程只允许附加一个取消边界事件。
- 如果事务子流程中嵌套了子流程，仅仅触发已经完成了的子流程的补偿事件。
- 对于多实例的事务子流程，如果其中一个实例触发了取消事件，那么其他的实例也同样会被触发取消边界事件。

 **注意** 取消边界事件不需要设置cancelActivity属性，因为它总是“中断”取消边界事件附加任务后续活动的执行。

补偿边界事件用于事务子流程（嵌入子流程不支持补偿边界事件）中针对事务失败后的业务逻辑进行补偿。4.5.4节中的图4-41很清楚地说明了补偿边界事件的作用。

补偿边界事件的图形化表示在中间事件的基础上嵌套了两个左三角符号。和其他的边界事件不同的是，补偿边界事件的输出流不是一个顺序流而是一个“关联”，用虚线表示。如图4-49展示了补偿边界事件及关联补偿活动，对应的XML描述如代码清单4-46所示。

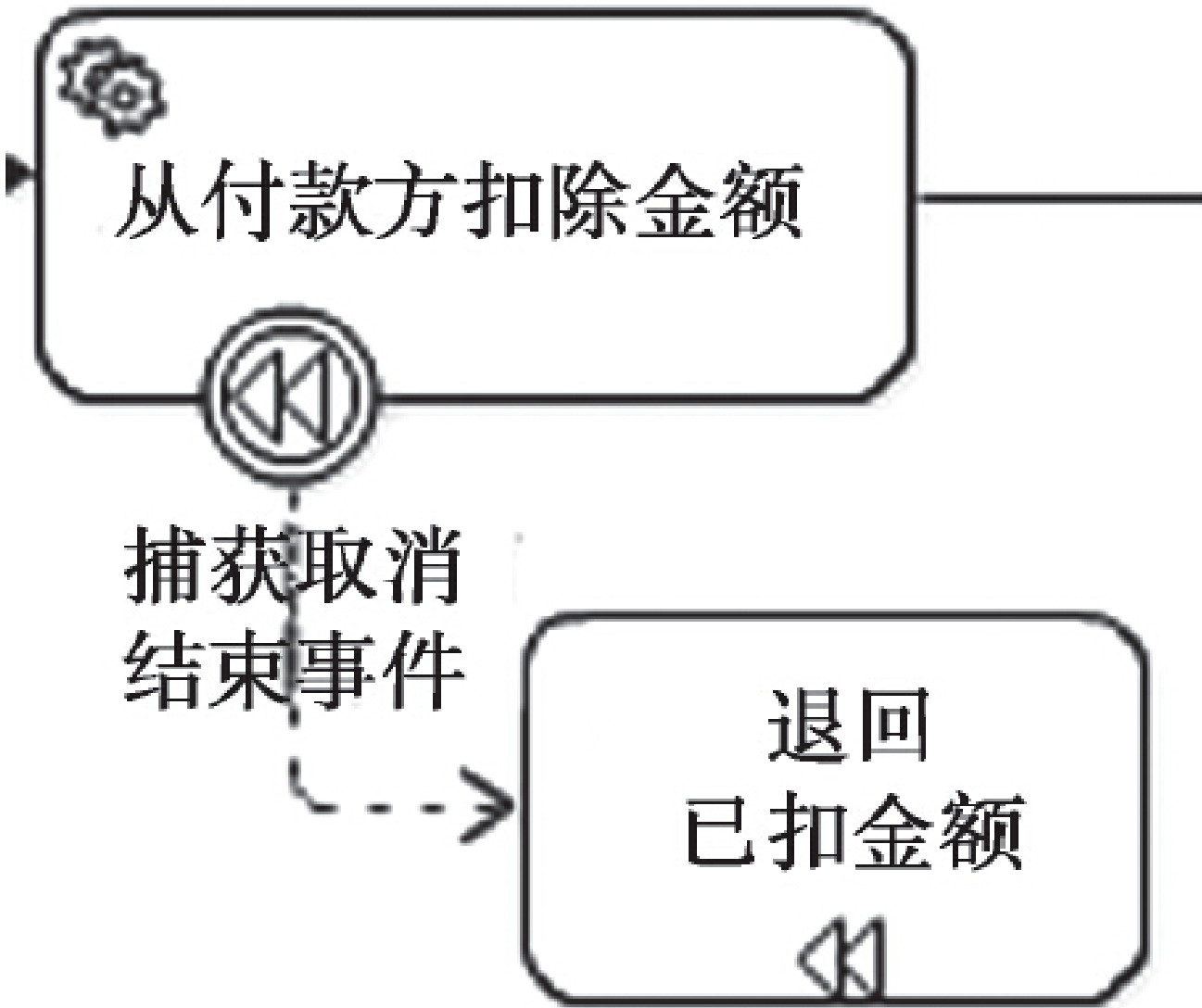


图4-49 补偿边界事件以及关联补偿活动

代码清单4-46 补偿边界事件的XML描述

```
<boundaryEvent id="Event_1" name="捕获取消结束事件" attachedToRef="Task_2">
  <compensateEventDefinition id="EventDef_1"/>
</boundaryEvent>
<association id="Association_1" sourceRef="Event_1"
  targetRef="Task_4" associationDirection="One"/>
<serviceTask id="Task_4" name="退回已扣金额" isForCompensation="true"/>
```

在代码清单4-46中，用`compensateEventDefinition`表示一个补偿事件；用`association`关联两个对象，和顺序流类似，使用`sourceRef`、`targetRef`表示两个关联的活动ID；在`serviceTask`中添加了`isForCompensation`属性，用来声明这是一个补偿类型的活动。

注意 如果补偿边界事件“附加”的活动是多实例（4.3.12节），当抛出补偿事件时，每一个实例都会被触发补偿边界事件。

4.6.2 中间捕获事件

中间捕获事件是流程中的“拦路虎”，根据事件的不同类型需要使用不同的方式才能继续执行后续的输出流的活动。例如执行完一个任务1天后执行下一个活动，或者一个指定的信号、消息被接收到之后再执行后续输出流的活动。

中间捕获事件在图形表现形式上和边界事件是有区别的：边界事件需要“附加”在一个活动上，而且不需要输入流；而中间捕获事件必须连接一个输入流和一个输出流，所以根据这个特性命名为中间捕获事件。另外，所有的中间捕获事件的图标都是空心。

中间捕获事件的基本XML描述如代码清单4-47所示。

代码清单4-47 中间捕获事件的基本XML描述

```
<intermediateCatchEvent id="event1">
  <xxxEventDefinition />
</intermediateCatchEvent >
```

1.定时器中间捕获事件

定时器中间捕获事件和定时启动事件（4.1.1节）、定时器边界事件（4.6.1节）的功能、配置参数的方式都类似，都在一个特定的时间或指定间隔多长时间之后被触发，从而执行输出流后面的活动。代码清单4-48列出了定时器中间捕获事件的XML描述。

代码清单4-48 定时器中间捕获事件的XML描述

```
<intermediateCatchEvent id="timerintermediatecatchevent1" name="TimerCatchEvent">
  <timerEventDefinition>
    <timeDuration>PT5M</timeDuration>
  </timerEventDefinition>
</intermediateCatchEvent>
```

定时器中间捕获事件的图形化表示和定时器边界事件一样，唯一的区分方式就是在流程中所处的位置，图4-50展示了定时器中间捕获事件的应用场景。

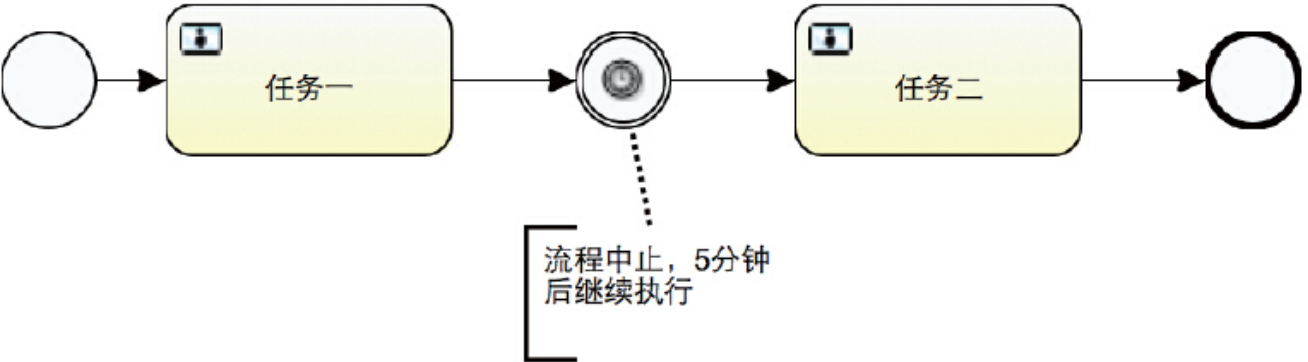


图4-50 定时器中间捕获事件的应用场景

定时中间捕获事件的XML的描述方式和定时器边界事件类似，不同的是由intermediateCatchEvent包裹timerEventDefinition。图4-51对应的XML描述如代码清单4-49所示。

代码清单4-49 图4-50对应的XML描述

```
<process id="process1" name="process1">
  <startEvent id="startevent1" name="Start"></startEvent>
  <userTask id="usertask1" name="任务一"></userTask>
  <sequenceFlow id="flow1" name="" sourceRef="startevent1" targetRef="usertask1"/>
  <intermediateCatchEvent id="timerintermediatecatchevent1" name="TimerCatchEvent"> #1-S
    <timerEventDefinition>
      <timeDuration>PT5M</timeDuration> #1-1
    </timerEventDefinition>
  </intermediateCatchEvent> #1-E
  <sequenceFlow id="flow2" name="五分钟后执行" sourceRef="usertask1" targetRef="timerintermediatecatchevent1"></sequenceFlow>
  ...
</process>
```

在代码清单4-49中，#1-S处定义了中间捕获事件，接着在#1-1处设置定时器在5分钟之后执行。关于定时器的不同方式可参考4.1.1节的“定时启动事件”。

2.信号中间捕获事件

信号中间捕获事件用来捕获被当前流程或其他流程抛出的信号事件，捕获的条件就是信号的ID一致，在4.4.4节的“事件网关”中其实已经涉及了信号中间捕获事件。

信号中间捕获事件的图形化表示是在中间事件的基础上嵌入一个空心三角符号，如图4-51所示，对应的XML描述如代码清单4-50所示。



图4-51 信号中间捕获事件

代码清单4-50 信号中间捕获事件的XML描述

```
<intermediateCatchEvent id="signal">  
  <signalEventDefinition signalRef="doXxxSignal" />  
</intermediateCatchEvent>
```

代码清单4-47在intermediateCatchEvent中嵌套了signalEventDefinition，并用signalRef属性定义一个信号的ID。

图4-52展示了一个信号中间捕获事件的应用场景，在“处理任务”的用户任务完成之后，流程处于“等待”状态，一直接收到指定的信号才能继续执行。

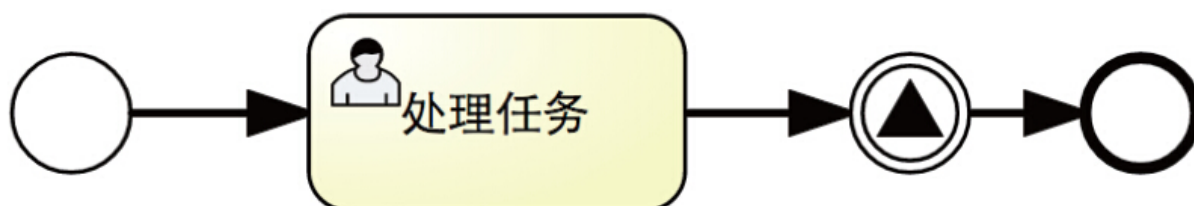


图4-52 信号中间捕获事件的应用场景

代码清单4-51列出了图4-52对应的XML描述。

代码清单4-51 图4-52对应的XML描述

```
<signal id="signal-001" name="发送通知信号"></signal> #1
<process id="process1" name="process1">
  <startEvent id="startevent1" name="Start"></startEvent>
  <sequenceFlow id="flow1" name="" sourceRef="startevent1" targetRef="usertask1"/>
  <intermediateCatchEvent id="signalintermediatecatchevent1" name="SignalCatchEvent"> #2
    <signalEventDefinition signalRef="signal-001"></signalEventDefinition> #3
  </intermediateCatchEvent>
  <sequenceFlow id="flow2" name="" sourceRef="usertask1"
    targetRef="signalintermediatecatchevent1"></sequenceFlow>
  <endEvent id="endevent1" name="End"></endEvent>
  <userTask id="usertask1" name="处理任务"></userTask>
  <sequenceFlow id="flow3" name="" sourceRef="signalintermediatecatchevent1"
    targetRef="endevent1"></sequenceFlow>
</process>
```

代码清单4-51在#1处使用signal定义了一个id为“signal-001”的信号，在#2处定义了信号中间捕获事件，并且在#3处设置信号的引用为“signal-001”。在第一个“处理任务”的用户任务完成之后通过输出流转到信号中间捕获事件，使流程处于等待状态，在有信号被抛出之后流程重新被激活，继续执行后续活动。

有一个地方需要读者注意，当抛出的信号事件匹配到多个相同的信号捕获事件时（可以是不同流程），每个匹配到的事件均会被执行，这也被称为“广播式”。

3.消息中间捕获事件

消息中间捕获事件和信号中间捕获事件类似，不同的是信号事件是“广播式”传播，而消息中间捕获事件是定向一对一的传递，也就是说一次只能把一个消息发给一个指定的流程实例。

消息中间捕获事件的图形化表示是在中间事件的基础上嵌入一个空心的信封图标，如图4-53所示，代码清单4-52列出了消息中间捕获事件的XML描述。



图4-53 消息中间捕获事件

代码清单4-52 消息中间捕获事件的XML描述

```
<intermediateCatchEvent id="messageCatch">
  <messageEventDefinition messageRef="continueXxx" />
</intermediateCatchEvent>
```

代码清单4-52在intermediateCatchEvent中嵌套了messageEventDefinition，并用messageRef属性定义一个消息名称。

图4-54展示了一个消息中间捕获事件的应用场景，在“处理任务”的用户任务完成之后，流程处于“等待”状态，一直接收到指定名称的消息才能继续执行。

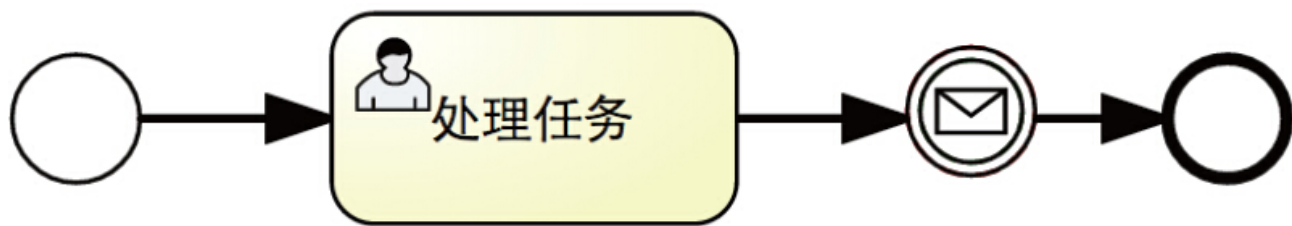


图4-54 消息中间捕获事件的应用场景

代码清单4-53 图4-55对应的XML描述

```
<message id="newInvoice" name="newInvoiceMessage" /> #1
<process id="process">
  <startEvent id="theStart" />
  <sequenceFlow id="flow1" sourceRef="theStart" targetRef="messageCatch" />
  <intermediateCatchEvent id="messageCatch"> #2
    <messageEventDefinition messageRef="newInvoice" /> #3
  </intermediateCatchEvent>
  <sequenceFlow id="flow2" sourceRef="messageCatch" targetRef="task" />
  <userTask id="task" name="处理任务" />
  <sequenceFlow id="flow3" sourceRef="task" targetRef="theEnd" />
  <endEvent id="theEnd" />
</process>
```

代码清单4-53在#1处使用message定义了一个id为“newInvoice”的消息，在#2定义了消息中间捕获事件，并且在#3处设置消息的引用为“newInvoice”。在第一个用户任务“处理任务”完成之后，通过输出流流转到消息中间捕获事件，使流程处于等待状态，当此流程实例收到消息之后，流程重新被激活继续执行后续活动（因为消息事件是一对一方式传播）。

4.6.3 中间抛出事件

中间抛出事件和中间捕获事件是两个相互依赖的关系，中间捕获事件需要有事件抛出才能被触发，而中间抛出事件需要有对应的捕获事件接收才有意义。

中间抛出事件一般用在一个任务完成后需要发送通知或执行其他系统任务的场景，工作流引擎会对抛出的事件进行传播（不同类型的事件有不同的作用范围）。

中间抛出事件的图形化表示和中间捕获事件正好相反，使用实心的图标而非中间捕获事件的空心图标。

中间抛出事件的基本XML表示如代码清单4-54所示。

代码清单4-54 中间抛出事件的基本XML描述

```
< intermediateThrowEvent id="event1">
  <xxxxEventDefinition />
</ intermediateThrowEvent >
```

1.空中间抛出事件

空中间抛出事件是BPMN 2.0规范中没有任何功能的事件，因此执行到空中间抛出事件时直接跳过。从业务层面对其理解，可以把它作为中间状态、结果的处理器，这就要借助Activiti的扩展功能来实现了，可以为它添加到监听器（4.7节）中执行我们预设的业务功能。

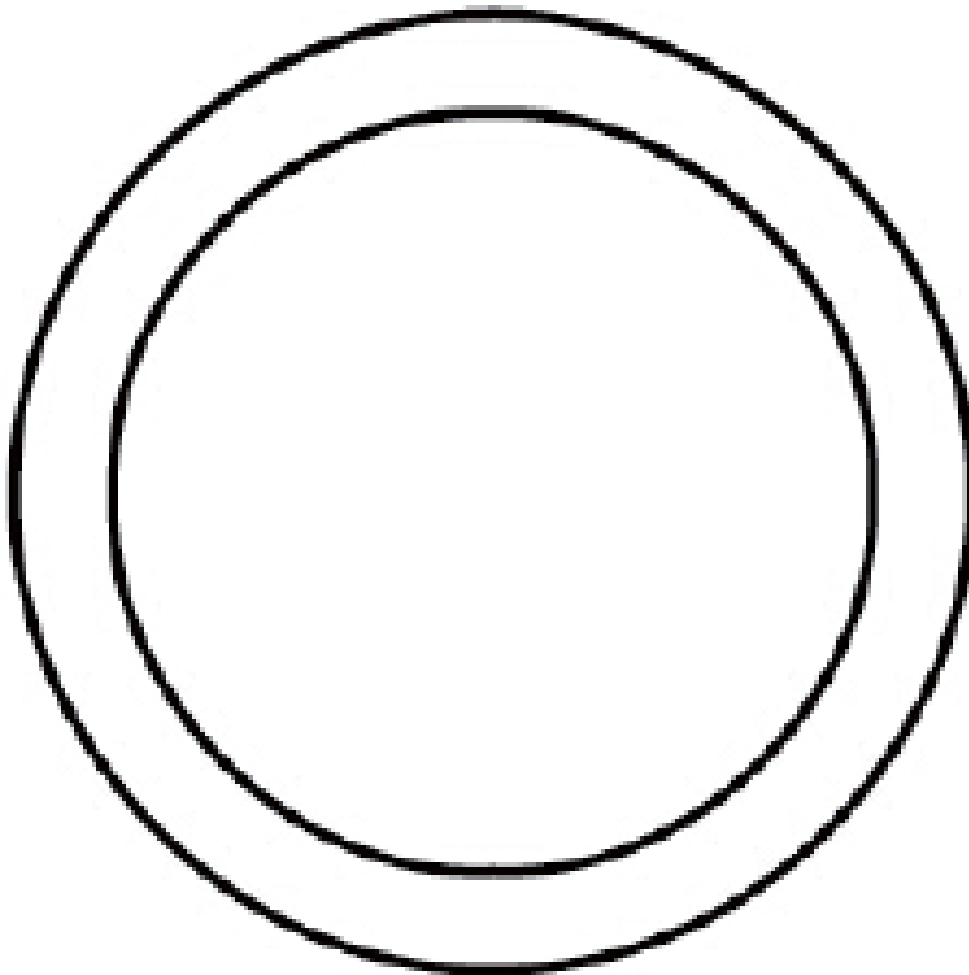


图4-55 空中间抛出事件

空中间抛出事件的图形化表示和中间事件的基础形状一样，仅仅只有一个双层圆形，如图4-55所示，对应的XML描述如代码清单4-55所示。

代码清单4-55 空中间抛出事件的XML描述

```
<intermediateThrowEvent id="noneintermediatethrowevent1" name="NoneThrowEvent">
</intermediateThrowEvent>
```

从代码清单4-55可以看出，仅仅使用一个intermediateThrowEvent标签即可描述一个空中间抛出事件，在标签内部不需要定义任何其他子元素。

读者可能会问：这样的事件有何作用呢？BPMN是用语言方式描述业务逻辑，明确地告知用户流程的流向及每一步的处理任务，空中间抛出事件虽然是一个空的任务，但是可以借助Activiti对大多数活动、事件添加的扩展功能使其更有意义，代码清单4-56为空中间抛出事件添加了一个监听器。

代码清单4-56 为空中间抛出事件添加监听器

```
<intermediateThrowEvent id="noneintermediatethrowevent1" name="NoneThrowEvent">
  <extensionElements>
    <activiti:executionListener #1
      class="me.kafeitu.listener.ChangeBusinessState"></activiti:executionListener> #2
    </extensionElements>
  </intermediateThrowEvent>
```

代码清单4-56在#1处为空中间抛出事件添加了Activiti的扩展功能activiti:execution-Listener，让空中间抛出事件更有意义，在#2处设置了监听的class名称。在实际应用过程中，可以在监听器中书写业务逻辑，当然流程变量也是可以获取的。

2.信号中间抛出事件

顾名思义，信号中间抛出事件可以抛出一个信号，然后交给引擎传播信号事件。

信号中间抛出事件的图形化表示是在中间事件的基础上嵌套一个实心的三角符号，如图4-56所示，对应的XML描述如代码清单4-57所示。



图4-56 信号中间抛出事件

代码清单4-57 信号中间抛出事件的XML描述

```
<intermediateThrowEvent id="noneintermediatethrowevent1" name=" SignalThrowEvent ">
  <signalEventDefinition signalRef="signal-001"></signalEventDefinition>
</intermediateThrowEvent>
```

图4-57展示了信号中间抛出事件的一个应用场景。

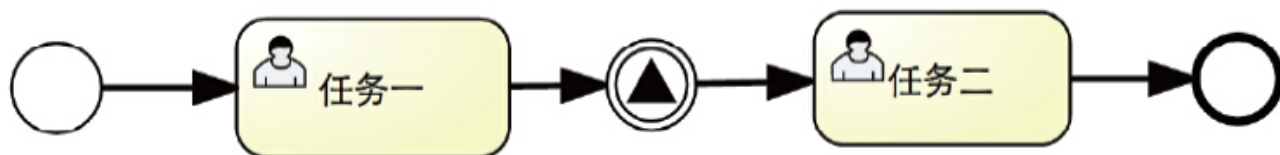


图4-57 信号中间抛出事件的一个应用场景

代码清单4-58列出了图4-57对应的XML描述。

代码清单4-58 图4-57对应的XML描述

```
<signal id="signal-001" name="发送通知信号"></signal> #1
<process id="SignalIntermediateThrowEvent" name="SignalIntermediateThrowEvent">
  <startEvent id="startevent1" name="Start"></startEvent>
  <userTask id="usertask1" name="任务一"></userTask>
  <sequenceFlow id="flow1" name="" sourceRef="startevent1" targetRef="usertask1"/>
  <intermediateThrowEvent id="siEvent1" name="SignalThrowEvent"> #2
    <signalEventDefinition signalRef="signal-001"></signalEventDefinition> #3
  </intermediateThrowEvent>
  <sequenceFlow id="flow2" name="" sourceRef="usertask1" targetRef=" siEvent1"/>
  <userTask id="usertask2" name="任务二"></userTask>
  <sequenceFlow id="flow3" name="" sourceRef=" siEvent1" targetRef="usertask2"/>
  <endEvent id="endevent1" name="End"></endEvent>
  <sequenceFlow id="flow4" name="" sourceRef="usertask2" targetRef="endevent1"/>
</process>
```

代码清单4-58在#1处首先定义了一个信号“signal-001”，在#2处定义了一个信号中间抛出事件，紧接着在#3处设置抛出的信号引用“signal-001”。在“任务一”执行完成之后直接抛出信号事件并等待匹配的信号捕获事件执行完成，如果在执行过程中有异常抛出，那么事务回滚，否则继续执行信号中间抛出事件的输出流，也就是创建“任务二”。

代码清单4-58的抛出是“同步”的，如果其中一个失败，那么其他的已经执行过的信号事件全部回滚；同样，也可以设置非同步即“异步”执行，意思就是引擎只会把信号抛出而不关心事务，在Activiti中会先创建和匹配到的信号事件相同数量的作业（Job，定时执行），由引擎负责作业的调度，不会等待其他信号事件的执行结果而继续执行信号中间抛出事件的输出流。

代码清单4-59列出了信号中间抛出事件的异步方式的XML描述。

代码清单4-59 信号中间抛出事件的异步方式的XML描述

```
<intermediateThrowEvent id="siEvent1" name="SignalThrowEvent">
  <signalEventDefinition signalRef="signal-001" activiti:async="true">
  </signalEventDefinition>
</intermediateThrowEvent>
```

从代码清单4-59可以看出，比代码清单4-58的#3处多了一个Activiti的扩展属性`activiti:async="true"`，这样即可表示一个异步的信号中间抛出事件。

4.7 监听器

监听器是Activiti在BPMN 2.0规范基础上扩展的功能，是业务与流程的“非侵入性粘合剂”。熟悉设计模式的读者应该能很快联想到“观察者模式”的一种实现“监听器模式”，在Activiti中开发人员可以通过配置监听器的方式监听各种动作，例如流程启动、结束、任务创建、任务完成，甚至是经过某个顺序流时。

监听器分为两类：执行监听器和任务监听器，和其他的Activiti扩展模型一样，监听器需要包含在BPMN 2.0规范的<extensionElements>标签中。

4.7.1 执行监听器

执行监听器允许在执行流程过程中执行Java代码（实现了监听器接口）或表达式。

执行监听器可以捕获的事件如下：

- 流程实例启动、结束
- 输出流捕获
- 活动启动、结束
- 路由开始、结束
- 中间事件开始、结束
- 触发开始事件、触发结束事件

多个事件的XML定义方式有一个共同点，都是在对应的监听对象内的extension-Elements中添加`activiti:executionListener`扩展属性。

执行监听器使用Activiti的扩展元素<activiti:executionListener>定义，通过event属性指定监听事件的类型（分为三类：start、end、take），并且可以选择监听器执行类型。表4-14列出了执行监听器的3种监听器执行类型。

表4-14 执行监听器的3种监听器执行类型

监听器执行类型	属性说明及示例
class	需要实现接口：org.activiti.engine.delegate.ExecutionListener <activiti:executionListener event="start" class="me.kafeitu.listener.ExecutionListenerForStart" />
expression	定义一个表达式，类似 EL 的语法。 <activiti:executionListener expression="\${pojo.method(execution.eventName)}" event="end" /> 这里的 pojo 是一个 Bean 的名称（可以用 spring 代理），还可以在 expression 中通过计算一个表达式配置监听器的名称
delegateExpression	class 属性必须显示指定一个 class 的全路径，如 me.kafeitu.ExecutionListenerXxx。在有些情况下需要根据业务的不同动态指定一个接口实现，delegateExpression 允许指定一个实现了监听接口（参考 class 属性的限制）类的 name，具体的类可以在引擎中配置或由 spring 代理。 <activiti:executionListener event="start" delegateExpression="\${executionListenerBean}" /> 其中 \${executionListenerBean } 为一个 Bean 对象的名称

 注意 这3种监听器执行类型只能使用其中的一种。

监听输出流（包含经过顺序流、条件顺序流）中没有开始、结束事件类型，只有在经过输出流的时候被触发，如代码清单4-60所示。

代码清单4-60 监听输出流的take事件

```
<process id="executionListenersProcess">
  <userTask id="firstTask" />
  <sequenceFlow sourceRef="firstTask" targetRef="secondTask">
    <extensionElements>
      <activiti:executionListener event="take" #1
        class="me.kafeitu.listener.ExecutionListenerForFlow" /> #2
    </extensionElements>
  </sequenceFlow>
</process>
```

代码清单4-60的#1处在sequenceFlow中定义了take类型执行监听，当流程经过这个顺序流的时候将会被触发，也就是执行#2处的me.kafeitu.listener.ExecutionListener-ForFlow类。

此外还可以在监听中使用<activiti:field/>注入字段（参考4.3.6节），在流程运行时监听实现类可以通过接口中的对象获取字段的值，如代码清单4-61所示。

代码清单4-61 在执行监听器中注入字段

```
<process id="executionListenersProcess">
  <userTask id="firstTask" />
  <sequenceFlow sourceRef="firstTask" targetRef="secondTask">
    <extensionElements>
      <activiti:executionListener event="take"
        class="me.kafeitu.listener.ExecutionListenerForFlow" >
        <activiti:field name="firstVar" stringValue="Henry Yan" /> #1
        <activiti:field name="secondVar" expression="${'user' + counter}" /> #2
      </activiti:executionListener>
    </extensionElements>
  </sequenceFlow>
</process>
```

代码清单4-61在#1处使用activiti:field为监听器me.kafeitu.listener.ExecutionListener-ForFlow注入了名称为firstVar的变量，这样在监听器中就可以通过调用Activiti引擎的方法获取变量的值；在#2处通过计算表单式user+counter的值为变量secondVar赋值，这样可以根据业务的状态动态设置变量的值。

4.7.2 任务监听器

相对于执行监听器的使用范围来说，任务监听器的使用范围要小很多，因为它只能应用于用户任务（4.3.1节），用来监听3种事件。

- **create**：在任务被创建且所有的任务属性设置完成之后才触发。
- **assignment**：在任务被分配给某个办理人之后触发。有一点需要注意，assignment事件总是在create事件触发之前被触发，因为任务的办理人是一个属性，而create事件需要逐一处理任务的办理人、候选人、候选组属性。
- **complete**：在配置了监听器的上一个任务完成时触发，也就是在运行时任务数据被删除之前（Activiti的数据分为运行时和历史两种类别，当运行时数据处理完毕后将会被清理）触发。

表4-15列出了任务监听器的3种监听器执行类型。

表4-15 任务监听器的3种监听器执行类型

监听器执行类型	属性说明及示例
class	需要实现接口：org.activiti.engine.delegate.TaskListener
expression	定义一个表达式，类似 EL 的语法。 <activiti:taskListener expression="\${pojo.method(execution.eventName)}" event="end" /> 这里的 pojo 是一个 Bean 的名称（可以用 spring 代理），还可以在 expression 中通过计算一个表达式配置监听器的名称
delegateExpression	class 属性必须显示指定一个 class 的全路径，如 me.kafeitu.TaskListenerOnXxxEvent。在有些情况下需要根据业务的不同动态指定一个接口实现，delegateExpression 允许指定一个实现了监听接口（参考 class 属性的限制）类的 name，具体的类可以在引擎中配置或由 spring 代理。 <activiti:taskListener event="start" delegateExpression="\${taskListenerBean}" /> 其中 \${ taskListenerBean } 为一个 Bean 对象的名称

代码清单4-62列出了请假流程中的“领导审批”节点完成时触发me.kafeitu.listener.TaskListenerForLeaveOnComplete监听实现类。

代码清单4-62 任务监听器的XML描述

```
<userTask id="leaderAudit" name="销假" >
  <extensionElements>
    <activiti:taskListener event="complete"
      class="me.kafeitu.listener.TaskListenerForLeaveOnComplete" />
  </extensionElements>
</userTask>
```

代码清单4-62在userTask中添加了activiti:taskListener元素来定义任务监听器，并且通过属性event=“complete”设置当任务完成时触发，从而执行监听器类me.kafeitu.listener.TaskListenerForLeaveOnComplett。

任务监听器和执行监听器一样可通过添加<activiti:field/>为任务监听器注入字段，参考代码清单4-61。

4.8 本章小结

本章讲解了Activiti目前支持的部分BPMN 2.0规范，是实战篇的理论基础。本章旨在从理论上使读者对BPMN 2.0规范及Activiti对其扩展上有一个全面的认识，了解BPMN 2.0规范能做什么，Activiti又在其基础上封装和扩展了什么。

本章每一节都是从概念开始引导，然后展示活动的基本XML描述，最后利用日常用例讲解各个规范的实际用途。

在下一部分——实战篇中将会以OA系统为例讲解如何使用Activiti作为工作流引擎开发OA系统的业务模块。当读者对于实战篇中的例子迷惑的时候，不妨再回顾一下规范对其是如何定义的。

第三部分 实战篇

经过了准备篇和基础篇的学习，大家了解了Activiti的大体架构，学会了如何运行一个Hello World程序，并且从概念和基础的XML定义上了解了BPMN 2.0规范及Activiti对其进行的一些扩展功能。

本篇在前面两篇内容的基础上正式进入Activiti的实战阶段，本篇的每个章节都会在日常流程基础上进行讲解，这样比较容易从业务层理解，也有利于学会如何应用Activiti的各种功能。本篇主要以大多数公司或者政府单位使用的OA系统为基础，一步一步讲解如何用Activiti进行构建。

在TDD盛行的今天我们当然也不能落下前进的脚步，TDD的好处一大堆，有兴趣的读者可以参考相关书籍深入学习。本书大部分代码的展示会以单元测试为主，这样既可保证代码的可执行性，又有利于读者对整个过程的理解。对于没有接触过TDD的读者也不用太担心，在实例代码中可以一步一步解释其含义。

第5章 用户与组及部署管理

把本章作为实战部分的第一章是因为用户和流程部署的管理是一把金钥匙。在工作流中最重要的参与者是人，流程中定义了何时需要人参与、何时由系统处理，所以在开始实例之前首先要学会管理这些基础数据以及它们之间的关系。

首先讲解Activiti中内置的一套用户、组的关系，以及如何通过API添加、删除、查询；然后就是流程定义的管理，包括部署、删除、读取部署的资源文件，还有中国用户比较关注的流程图中文乱码问题的解决办法。

通过本章内容读者将学会如何通过调用Activiti的API来管理用户以及流程定义。

 **说明** 本书的示例代码均使用Maven构建，对于一些配置读者可参考bpmn20-example项目中的pom.xml文件，其中用中文详细说明了各个配置的作用。

5.1 用户与组

Activiti中内置了一套相对简单的对用户和组的支持，可以满足基本业务需求对用户和组的要求。其中，组也可以理解为我们常说的角色，和用户的关系是多对多。相信这个关系读者应该不陌生了，可以回顾一下自己公司或项目中的用户组织权限的结构。

在学习过程中除了使用单元测试来帮助理解之外，读者还可以结合2.5节中的Activiti Explorer实际操作来加深理解。

5.1.1 用户

在各种需要人工参与的系统中，用户和组是一个身份系统（或者模块）的基础，在Activiti中用户和组主要是应用于用户任务（userTask，见4.3.1节）。

代码清单5-1利用单元测试说明操作Activiti中用户管理功能的各个API的使用。

代码清单5-1 用户管理的API演示

```
public class IdentifyServiceTest {
    @Rule
    // 使用默认的activiti.cfg.xml作为参数
    public ActivitiRule activitiRule = new ActivitiRule();           #1
    // 用户管理的API演示
    @Test
    public void testUser() throws Exception {
        // 获取IdentifyService实例
        IdentifyService identifyService = activitiRule.getIdentityService();           #2
        // 创建一个用户对象
        User user = identifyService.newUser("henryyan");           #3-S
        user.setFirstName("Henry");
        user.setLastName("Yan");
        user.setEmail("yanhonglei@gmail.com");           #3-E
        // 保存用户到数据库
        identifyService.saveUser(user);           #4
        // 验证用户是否保存成功
        User userInDb = identifyService.createUserQuery()           #5-S
            .userId("henryyan").singleResult();           #5-E
        assertNotNull(userInDb);
        // 删除用户
        identifyService.deleteUser("henryyan");           #6
        // 验证是否删除成功
        userInDb = identifyService.createUserQuery().userId("henryyan").singleResult();           #7-S
        assertNull(userInDb);           #7-E
    }
}
```

从代码清单5-1中可以看出，这是一个日常开发经常接触的CRUD操作，使用方式都比较简单，而且支持链式操作，读者可以在解压activiti-version.zip后在docs/javadoc中找到API文档，在1.3节的“引擎API”中提到的7个Service接口都位于org.activiti.engine包中。

下面——讲解每一行代码的作用。

#1处的作用是创建一个测试的Rule对象，并且使用指定的“activiti.cfg.xml”作为配置文件；当创建ActivitiRule对象时会自动创建一个引擎对象ProcessEngine，这样就可以通过activitiRule实例获取7个Service及其他对象了。此处也可以不指定“activiti.cfg.xml”参数，即通过new ActivitiRule()一样可以创建一个基于内存数据库（H2）的引擎实例，这是因为ActivitiRule默认使用名称为“activiti.cfg.xml”的配置文件，读者可以修改代码进行验证。

#2处就比较简单了，通过ActivitiRule对象获取一个IdentityService对象的实例。细心的读者可能发现这里获取实例的方法与第2章中的方式不同，但是结果是一样的，因为Activiti提供了多种不同的方式来获取各个Service实例。在第7章中还会介绍通过Spring注入Service实例的方式。

#3处用于创建一个ID为“henryyan”的User对象，但是并不会立即保存到数据库，而是创建一个临时对象，紧接着设置了用户的其他属性。

#4处保存一个User对象，只有调用了saveUser才会把创建或更改的最新属性持久化到数据库。

#5处用于查询数据库中的用户数据，首先需要创建一个用户查询对象即调用“identityService.createUserQuery()”方法，接着使用“链式^[1]”编程方式设置用户的ID属性等于“henryyan”，最后调用list()方法返回一个对象集合。当然除了获取一个对象集合外，还可以只获取单个对象，把list()换成singleResult()即可，当然对应的返回类型就不能为List了，而是User。

#6处通过指定一个User的ID属性值删除一个User对象。

#7处和#5处相当于验证组的数量是否和期望一致，在#6处已经删除了User对象，所以这里查询的用户对象为空。

5.1.2 组

组是控制权限的一种方式，属于某个组的用户就拥有操作某些功能的权限。本节详细说明Activiti内置的组管理及API的使用。

在Activiti中，组的类型分为两种，即assignment和security-role，前者为一种普通的岗位角色，是用户分配业务中的功能权限，后者是安全角色，可以从全局管理用户组织及整个流程的状态。

代码清单5-2利用单元测试讲解操作Activiti中组管理功能的各个API的使用。

代码清单5-2 组管理的API演示

```
public class IdentityService Test {
    @Rule
    public ActivitiRule activitiRule = new ActivitiRule("activiti.cfg.xml");
    // 组管理的API演示
    @Test
    public void testGroup() throws Exception {
        // 获取IdentifyService实例
        IdentityService identityService = activitiRule.getIdentityService();
        // 创建一个组对象
        Group group = identityService.newGroup("deptLeader");
        group.setName("部门领导");
        group.setType("assignment");
        // 保存组
        identityService.saveGroup(group);
        // 验证组是否已保存成功，首先需要创建组查询对象
        List<Group> groupList = identityService.createGroupQuery().groupId("deptLeader").list();
        assertEquals(1, groupList.size());
        // 删除组
        identityService.deleteGroup("deptLeader");
        // 验证是否删除成功
        groupList = identityService.createGroupQuery().groupId("deptLeader").list();
        assertEquals(0, groupList.size());
    }
}
```

代码清单5-2和代码清单5-1类似，只不过把用户换成了组，所以不再——介绍每一步的功能，读者可以查看bpmn20-example中的IdentityServiceTest类运行单元测试的验证结果。

5.1.3 用户与组的关系

在5.2.1和5.2.2节中分别介绍了如何使用Activiti的API操作用户和组。如果仅仅只有用户和组，那只能说是一堆散沙，本节将介绍如何让这两者建立起对象联关系。

代码清单5-3演示了如何创建用户与组的关系，以及如何根据用户或组查询关联的另外一方。

代码清单5-3 设置用户与组的关系

```
public class IdentifyServiceTest {
    @Rule
    public ActivitiRule activitiRule = new ActivitiRule("activiti.cfg.xml");
    // 组和用户关联关系API演示
    @Test
    public void testUserAndGroupMembership() throws Exception {
        // 创建并保存组对象
        Group group = identityService.newGroup("deptLeader");          #1-S
        group.setName("部门领导");
        group.setType("assignment");
        identityService.saveGroup(group);          #1-E
        // 创建并保存用户对象
        User user = identityService.newUser("henryyan");          #2-S
        user.setFirstName("Henry");
        user.setLastName("Yan");
        user.setEmail("yanhonglei@gmail.com");
        identityService.saveUser(user);          #2-E
        // 把用户henryyan加入到组deptLeader中
        identityService.createMembership("henryyan", "deptLeader");          #3
        // 查询属于组deptLeader的用户
        User userInGroup =          #4-S
        identityService.createUserQuery().memberOfGroup("deptLeader").singleResult();
        assertNotNull(userInGroup);
        assertEquals("henryyan", userInGroup.getId());          #4-E
        // 查询henryyan所属组
        Group groupContainsHenryyan =          #5-S
        identityService.createGroupQuery().groupMember("henryyan").singleResult();
        assertNotNull(groupContainsHenryyan);
        assertEquals("deptLeader", groupContainsHenryyan.getId());          #5-E
    }
}
```

在代码清单5-3中，首先创建组、用户，然后设置两者的关系，最后以不同的方式查询关联的对方。

#1处的作用在5.1.2已经介绍了，用来创建并保存一个组对象。

#2处的作用也在5.1.1绍了，用来创建并保存一个用户对象。

#3处是本例的关键代码，通过调用方法createMembership(“henryyan”，“deptLeader”)把用户“henryyan”加入到组“deptLeader”中。

#4处用memberOfGroup()方法查询属于组“deptLeader”的用户，此处只保存了一个用户，所以用sigleResult()方法返回一个用户实例，如果保存了多个用户，就需要使用list()方法代替。

#5处和#4相反，用groupMember()方法从组的这一方查询包含的用户

5.1.4 用户任务中的用户与组

有了用户、组，以及两者之间关系的基础，再来看看如何在实际应用中使用它们，这里通过一个只有一个用户任务节点的简单流程进行演示。

1.候选组

首先来看一下这个例子的流程图，如图5-1所示。

在学习本节的示例之前先回顾一下2.4.4节中提到的“签收”（代码清单2-5），当不确定要将一个任务分配给哪个人办理时，就需要把任务指定给多个候选人或候选组。本节的示例是把任务分配给一个候选组“deptLeader”，意思就是属于这个组的用户都可以是“潜在”的办理人，但是大家都遵守一个规则：谁“签收”了这个任务谁就可以办理任务。

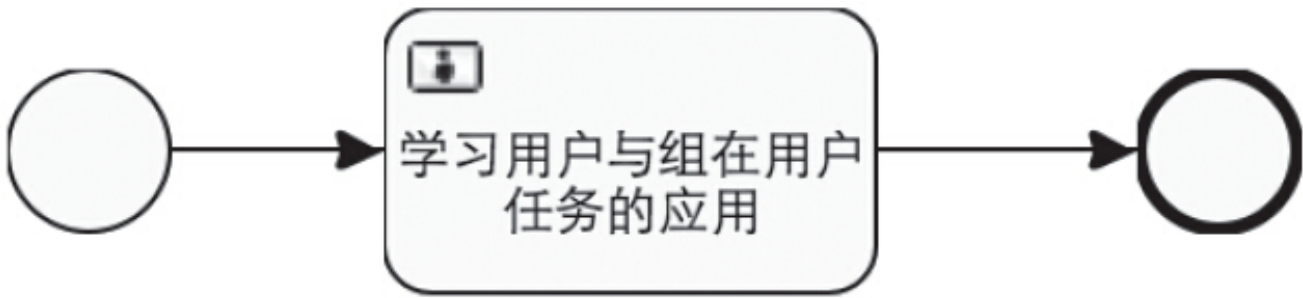


图5-1 学习用户与组在用户任务中的应用流程图

首先浏览一下图5-1对应的XML代码，如代码清单5-4所示。

代码清单5-4 图5-1对应的XML

```
<process id="userAndGroupInUserTask" name="userAndGroupInUserTask"> #1
  <startEvent id="startevent1" name="Start"></startEvent>
  <userTask id="studyUserAndGroupInUserTask" #2
    name="学习用户与组在用户任务中的应用"
    activiti:candidateGroups="deptLeader"></userTask> #3
  <sequenceFlow id="flow1" name="" sourceRef="startevent1"
    targetRef="studyUserAndGroupInUserTask"></sequenceFlow>
  <endEvent id="endevent1" name="End"></endEvent>
  <sequenceFlow id="flow2" name="" sourceRef="studyUserAndGroupInUserTask"
    targetRef="endevent1"></sequenceFlow>
</process>
```

代码清单5-4的#1处定义了一个ID为“userAndGroupInUserTask”的流程定义。

#2处定义了一个用户任务，对应图5-1中的“学习用户与组在用户任务中的应用”部分。

#3处用4.3.1节介绍过的activiti:candidateGroups属性定义用户任务的候选组为“deptLeader”。

了解了这个流程定义的内容后，接着通过测试代码驱动方式学习用户、组及它们之间的关系如何在用户任务中应用。

有了5.1.1、5.1.2、5.1.3小节的基础之后我们对新的单元测试稍微调整，把添加用户、组及建立关系的代码抽取成一个方法，见代码清单5-5。

代码清单5-5 单元测试中封装了用户、组的初始化及清理方法

```
public class UserAndGroupInUserTaskTest extends AbstractTest {
    @Before
    public void setUp() throws Exception {
        // 初始化7个Service实例
        super.setUp();
        // 创建并保存组对象
        Group group = identityService.newGroup("deptLeader");
        group.setName("部门领导");
        group.setType("assignment");
        identityService.saveGroup(group);
        // 创建并保存用户对象
        User user = identityService.newUser("henryyan");
        user.setFirstName("Henry");
        user.setLastName("Yan");
        user.setEmail("yanhonglei@gmail.com");
        identityService.saveUser(user);
        // 把用户henryyan加入到组deptLeader中
        identityService.createMembership("henryyan", "deptLeader");
    }
    @After // 每个方法执行完后清理用户与组
    public void afterInvokeTestMethod() throws Exception {
        identityService.deleteMembership("henryyan", "deptLeader");
        identityService.deleteGroup("deptLeader");
        identityService.deleteUser("henryyan");
    }
}
```

这里针对不太了解JUnit的读者稍微做一下介绍，在方法setUp()上注解@Before的作用：当测试（有@Test注解的方法）方法开始执行之前会被调用一次；相反在afterInvoke-TestMethod()方法上@After注解的作用：当一个测试方法执行结束之后执行一次。

本示例的作用就是在每个方法执行之前添加用户、组并建立关系，每个测试方法执行结束之后再清理添加的数据。

基于前面的介绍来分析一段有关用户任务中候选组的代码片段，见代码清单5-6。

代码清单5-6 用户任务中的候选组

```
@Test
```

```

@Deployment(resources = { "chapter5/userAndGroupInUserTask.bpmn" })
public void testUserAndGroupInUserTask() throws Exception {
    // 启动流程
    ProcessInstance processInstance =
        runtimeService.startProcessInstanceByKey("userAndGroupInUserTask");    #1
    assertNotNull(processInstance);
    // 根据角色查询任务
    Task task = taskService.createTaskQuery().taskCandidateUser("henryyan").singleResult();    #2
    taskService.claim(task.getId(), "henryyan");    #3
    taskService.complete(task.getId());    #4
}

```

代码清单5-6的目的就是读取用户“henryyan”的候选任务，下面——分析。

#1处大家已经很熟悉了，根据流程定义的ID启动一个流程实例。

#2处用于查询用户任务，并且通过条件taskCandidateUser("henryyan")过滤包含候选人“henryyan”的用户任务。

解释一下“候选人”的含义，回顾4.3.1节对用户任务的介绍，其中提及两个属性：activiti.cadidateUsers和activiti.cadidateGroups。如果在属性activiti.cadidateUsers中设置多个用户，那么这些用户都属于“候选人”；activiti.cadidateGroups是定义候选组。候选人与候选组的不同在于后者可以包含前者，所有属于候选组中的用户都可以理解为“候选人”。下面通过图5-2更好地了解两者之间的关系。



图5-2 描述“候选人”概念的图形方式

代码清单5-6用示例说明了用户、组在用户任务中的使用，并且通过图5-2直观地说明了候选人的含义。在此基础上再来看看如果一个组包含多个人在用户任务中该如何处理，对应的JUnit单元测试代码见代码清单5-7。

代码清单5-7 用户组中包含多个用户时在用户任务中的应用

```

@Test
@Deployment(resources = { "chapter5/userAndGroupInUserTask.bpmn" })
public void testUserTaskWithGroupContainsTwoUser() throws Exception {
    // 在setUp()的基础上再添加一个用户，然后加入到组deptLeader中
    User user = identityService.newUser("jackchen");    #1-S
    user.setFirstName("Jack");
    user.setLastName("Chen");
    user.setEmail("jackchen@gmail.com");
    identityService.saveUser(user);
    // 把用户henryyan加入到组deptLeader中
    identityService.createMembership("jackchen", "deptLeader");    #1-E
    // 启动流程
    ProcessInstance processInstance =
        runtimeService.startProcessInstanceByKey("userAndGroupInUserTask");
    assertNotNull(processInstance);
    // jackchen作为候选人的任务
    Task jackchenTask = taskService.createTaskQuery()    #2-S
        .taskCandidateUser("jackchen").singleResult();
    assertNotNull(jackchenTask);    #2-E
    // henryyan作为候选人的任务
    Task henryyanTask = taskService.createTaskQuery()    #3-S
        .taskCandidateUser("henryyan").singleResult();
    assertNotNull(henryyanTask);    #3-E
    // jackchen签收任务
    taskService.claim(jackchenTask.getId(), "jackchen");    #4
    // 再次查询用户henryyan是否拥有刚刚的候选任务
    henryyanTask = taskService.createTaskQuery()    #5-S
        .taskCandidateUser("henryyan").singleResult();
    assertNull(henryyanTask);    #5-E
}

```

代码清单5-7展示了用户组中包含多个用户时在用户任务中的应用。本示例想要说明的是：在一个用户任务被其中一个候选人签收之后，其他候选人同时失去拥有这个任务的权限。下面结合代码说明每个功能点的含义。

#1处是在setUp()方法基础上再添加一个用户“jackchen”到组“deptLeader”，此时组deptLeader就有两个用户，通过图5-2的解释可知，这两个用户都属于任务“studyUser-AndGroupInUserTask”的候选人。

- #2处验证用户“jackchen”作为候选人查询到一个任务。
- #3处验证用户“henryyan”作为候选人查询到一个任务。
- #4处用户“jackchen”签收任务“studyUserAndGroupInUserTask”，此时任务属于“jackchen”，其他候选人失去拥有这个任务的权限。
- #5处再次查询用户“henryyan”作为候选人的任务，查询结果应该为空，因为任务“studyUserAndGroupInUserTask”已经被用户“jackchen”签收，所以#5-E处的代码执行结果为true。

2.候选人

在代码清单5-6和代码清单5-7中都把一个组作为用户任务的候选组（包含多个候选人组的名称），如果多个候选人不属于任何候选组，那么是否可以把用户任务直接分配给多个候选人呢？答案当然是肯定的，继续通过一个例子来讲解把多个用户直接作为用户任务的候选人。

首先浏览一下代码清单5-8的流程定义文件。

代码清单5-8 用户任务中包含多个直接候选人

```
<process id="candidateUserInUserTask" name="candidateUserInUserTask">
  <startEvent id="startevent1" name="Start"></startEvent>
  <userTask id="usertask1" name="用户任务包含多个直接候选人"
    activiti:candidateUsers="jackchen, henryyan"></userTask> #1
  <sequenceFlow id="flow1" sourceRef="startevent1" targetRef="usertask1"/>
  <endEvent id="endevent1" name="End"></endEvent>
  <sequenceFlow id="flow2" sourceRef="usertask1" targetRef="endevent1"/>
</process>
```

代码清单5-8和代码清单5-4基本一致，唯一不同的是：在代码清单5-8的#1处把原来的“`activiti:candidateGroups`”换成了“`activiti:candidateUsers`”，并且设置其值为“jackchen, henryyan”，表示此任务有两个候选人。根据对本节其他例子的理解可以判定：在其中一个用户“签收”了任务之后其他的候选人就失效了。如果不理解属性`activiti:candidateUsers`，可以回顾一下4.3.1节表4-3中的解释。

代码清单5-9将验证我们对用户任务中候选人的期望。

代码清单5-9 在测试用户任务中直接设置多个候选人

```
public class CandidateUserInUserTaskTest extends AbstractTest {
    @Test
    @Deployment(resources = { "chapter5/candidateUserInUserTask.bpmn" })
    public void testMultiCadiateUserInUserTask() throws Exception {
        // 添加用户jackchen
        User userJackChen = identityService.newUser("jackchen");
        ...
        identityService.saveUser(userJackChen);
        // 添加用户henryyan
        User userHenryyan = identityService.newUser("henryyan");
        ...
        identityService.saveUser(userHenryyan);
        // 启动流程
        ProcessInstance processInstance =
            runtimeService.startProcessInstanceByKey("candidateUserInUserTask");
        assertNotNull(processInstance);
        // jackchen作为候选人的任务
        Task jackchenTask = taskService.createTaskQuery() #1-S
            .taskCandidateUser("jackchen").singleResult();
        assertNotNull(jackchenTask); #1-E
        // henryyan作为候选人的任务
        Task henryyanTask = taskService.createTaskQuery() #2-S
            .taskCandidateUser("henryyan").singleResult(); #2-E
        assertNotNull(henryyanTask);
        // jackchen签收任务
        taskService.claim(jackchenTask.getId(), "jackchen"); #3
        // 再次查询用户henryyan是否拥有刚刚的候选任务
        henryyanTask = taskService.createTaskQuery() #4-S
            .taskCandidateUser("henryyan").singleResult();
        assertNull(henryyanTask); #4-E
    }
}
```

正如我们期待的那样，代码清单5-9的执行结果是正确的，一切和我们设想的一样（单元测试真不愧是学习的利器，可以反复验证期望的结果直到结果为绿色）。

代码清单5-9中的#1、#2、#3、#4和代码清单5-7中的这几处完全一样，而且执行结果也是一样，这也再次证明了图5-2的说明。

关于用户、组及其在用户任务中的应用的介绍暂时告一段落，在稍后的章节还会通过更多的示例讲解更深层次的应用。

[1] 链式编程的表现形式为多个方法以“.”分割：在调用并执行完一个方法之后该方法返回当前方法的对象实例，这样可以继续调用返回对象实例的其他方法。链式编程可以减少临时变量的产生，并且可以让代码更加优雅，所以在各种框架和组件中经常被使用。在Activiti中所有的Query结尾的类都支持链式编程。

5.2 部署流程资源

流程资源可以是各种类型的文件，在启动流程或流程实例运行过程中会被读取。下面介绍常用的流程资源。

- 流程定义文件：扩展名为**bpmn20.xml**和**bpmn**；
- 流程定义的图片：用BPMN 2.0规范的各种图形描绘，一般用**PNG**格式；
- 表单文件：把表单内容保存在一个文件中，扩展名为**form**；
- 规则文件：例如Drools的规则文件，其扩展名为**drl**。

使用流程引擎的目的是为了使用流程驱动业务的流转。要用流程驱动业务就必须为业务启动一个流程实例，启动一个流程实例必须定义一系列活动，这一系列的活动就组成了一个流程定义。在之前的多个流程相关的例子中，总是先部署一个流程定义才能根据已部署的流程定义启动流程实例，但是仅仅使用了一种部署流程定义的方式。本节将详细讲解其他的部署方式及如何读取部署的资源文件。

部署流程资源有很多种方法，包括：classpath、InputStream、字符串、zip格式压缩包。之前的例子只涉及通过classpath方式部署流程定义，下面一一讲解各个部署方式。

虽然部署方式不同，但是均通过RepositoryService的createDeployment()方法在创建DeploymentBuilder对象后调用不同的方法部署资源。

在本节的最后用图形方式展示了部署的过程，作为参考帮助读者理解下面的代码。

5.2.1 classpath方式

classpath方式，顾名思义就是要以class目录为基础寻找对应的资源再部署，例如2.4节的Hello World。本章的代码清单5-6其实也是通过classpath方式部署流程定义文件的，只不过Activiti提供了CDI方式简化了部署，以便读者把注意力集中在测试代码。

classpath方式的路径名规范是用斜杠“/”方式分割多个包名。图5-3展示了一个流程定义文件的路径。

图5-3对应的classpath为chapter5/candidateUserInUser-Task.bpmn。

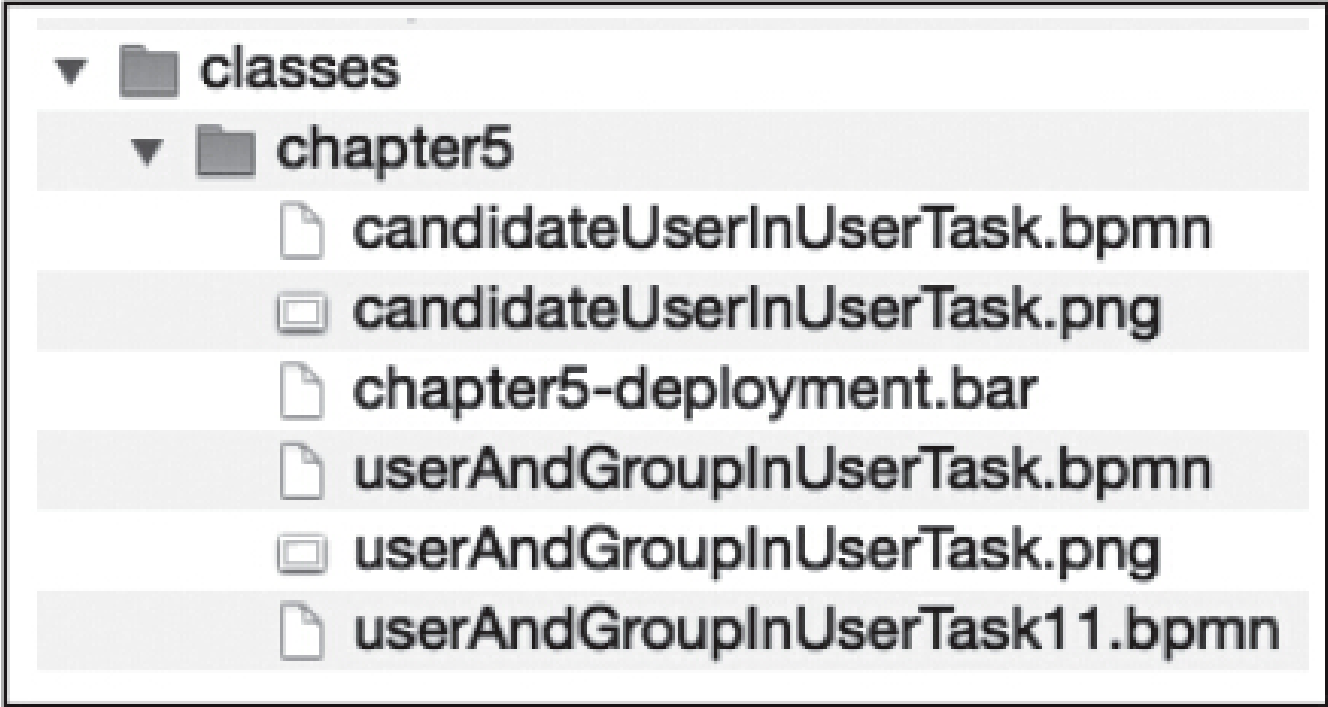


图5-3 class目录中的流程定义文件目录结构

代码清单5-10没有使用链式编程方式，而是列出了部署流程定义需要执行的几步。

代码清单5-10 用classpath方式部署流程定义文件

```
public class ClasspathDeploymentTest extends AbstractTest {
    @Test
    public void testClasspathDeployment() throws Exception {
        // 定义classpath
        String bpmnClasspath = "chapter5/candidateUserInUserTask.bpmn";
        String pngClasspath = "chapter5/candidateUserInUserTask.png";
        // 创建部署构建器
        DeploymentBuilder deploymentBuilder = repositoryService.createDeployment(); #1
        // 添加资源
        deploymentBuilder.addClasspathResource(bpmnClasspath); #2-S
        deploymentBuilder.addClasspathResource(pngClasspath); #2-E
        // 执行部署
        deploymentBuilder.deploy(); #3
        // 验证是否部署成功
        ProcessDefinitionQuery processDefinitionQuery = #4-S
            repositoryService.createProcessDefinitionQuery();
        long count = processDefinitionQuery.processDefinitionKey("candidateUserInUserTask")
            .count();
        assertEquals(1, count); #4-E
        // 读取图片文件
        ProcessDefinition processDefinition = processDefinitionQuery.singleResult(); #5-S
        String diagramResourceName = processDefinition.getDiagramResourceName(); #5-1
        assertEquals(pngClasspath, diagramResourceName); #5-E
    }
}
```

代码清单5-10是一个典型的classpath方式部署流程定义的方法，下面拆分了每一步的动作进行说明。

- #1处创建一个部署构建器，它是部署的入口。
- #2处添加了classpath资源，包括XML格式的bpmn文件和PNG格式的图片文件；该方法会立即读取classpath资源文件为一个输入流对象。
- #3处调用deploy()方法执行部署，此方法会把流程定义文件内容持久化到数据库，并且使用addClasspathResource(resourceName)方法的参数作为资源名称。
- #4处用于验证流程定义是否已经部署成功，根据流程定义的Key属性查询验证已经部署的流程定义数量应该为1。
- #5处用于验证流程定义对应的图片资源是否部署成功，在#5-S处得到一个流程定义对象，在#5-1处获取流程定义对应的图片资源的名称，在#5-E处验证图片资源名称是否是部署时的名称。

classpath方式一般用于开发阶段的测试环节，真正在产品环境中用得很少到，而是通过管理界面手动部署或在设计完流程之后直接部署到引擎中。

5.2.2 InputStream方式

使用InputStream方式部署流程资源需要传入一个输入流及资源的名称，输入流的来源不限，可以从classpath读取，可以从一个绝对路径文件读取，也可以是从网络上读取。

下面以代码来讲解如何通过InputStream方式部署流程资源，如代码清单5-11所示。

代码清单5-11 用InputSteam方式部署流程资源文件

```
public class InputStreamDeploymentTest extends AbstractTest {
    @Test // 从一个绝对文件流部署
    public void testInputStreamFromAbsolutePath() throws Exception {
        String filePath = "/Users/henryyan/work/books/aia-codes/bpmn20-example
        /src/test/resources/chapter5/userAndGroupInUserTask.bpmn"; #1
        // 读取classpath的资源为一个输入流
        FileInputStream fileInputStream = new FileInputStream(filePath); #2
        repositoryService.createDeployment() #3-S
        .addInputStream("userAndGroupInUserTask.bpmn", fileInputStream).deploy(); #3-E
        // 验证是否部署成功
        ProcessDefinitionQuery pdq = repositoryService.createProcessDefinitionQuery(); #4-S
        long count = pdq.processDefinitionKey("userAndGroupInUserTask").count(); #4-1
        assertEquals(1, count); #4-E
    }
}
```

代码清单5-11列出了用绝对路径的文件流作为InputStream对象部署到引擎。下面详细说明了每行代码的作用。

#1处定义了资源文件的绝对路径，读者在运行本例时可自行更改为自己环境的实际路径。

#2处创建了一个文件输入了对象。

#3处用“userAndGroupInUserTask.bpmn”作为资源名称，以fileInputStream为资源文件的输入流部署到引擎中。

#4处先（#4-S）创建一个流程定义查询对象，然后统计已部署流程的数量并验证数量是否为1。

InputStream方式在产品环境中用得比较多，例如从Web客户端接受一个文件对象部署到引擎中，或者从URL读取文件流后部署到引擎中。

5.2.3 字符串方式

利用字符串方式可以直接传入纯文本作为资源的来源。classpath方式和InputSteam方式都是读取一个输入流后部署资源，和这两种方式类似，字符串方式的实现原理是把一组字符串的内容转换为字节流后再部署。代码清单5-12列出了用字符串方式部署流程的代码。

代码清单5-12 用字符串方式部署流程资源文件

```
public class StringDeploymentTest extends AbstractTest {
    // XML字符串，为了节约版面省略了定义的内容，读者可参考示例的源码查看全部内容
    private String text = #1
        "<?xml version='1.0' encoding='UTF-8'><definitions>http://www.hzcourse.com/resource/readBook?path=/openresources/teach_ebook/uncompressed/15030/OEBPS/Text/...</d
    @Test
    public void testCharsDeployment() {
        // 以candidateUserInUserTask.bpmn为资源名称，以text的内容为资源内容部署到引擎
        repositoryService.createDeployment()
            .addString("candidateUserInUserTask.bpmn", text).deploy(); #2
        // 验证流程定义是否部署成功
        ProcessDefinitionQuery pdq = repositoryService.createProcessDefinitionQuery(); #3-S
        long count = pdq.processDefinitionKey("candidateUserInUserTask").count(); #3-1
        assertEquals(1, count); #3-E
    }
}
```

代码清单5-12的#1处把一组字符串赋值给text变量，不过此处把XML的内容省略了。读者可参考bpmn20-

example/src/text/java/me/kafeitu/activiti/chartper5/deployment/StringDeploymentTest.java类，其中的内容和candidateUserInUserTask.bpmn文件一样。

#2处用addString()方法设置资源名称为“candidateUserInUserTask.bpmn”，内容为text的变量，之后调用deploy部署，结果与用classpath和InputSteam一样。

#3处依旧验证部署是否成功。

5.2.4 zip/bar格式压缩包方式

以上3种部署方式一次只能部署一个资源，除非执行多次deployment.addXxx()方法，否则如何一次部署多个资源呢？很简单，Activiti允许把一批资源打包部署，例如在采用classpath方式部署的例子中同时部署了XML格式的“candidateUserInUserTask.bpmn”文件和图片格式的“candidateUserInUserTask.png”文件，用压缩包方式部署可以把这两个资源文件打包成一个扩展名为bar（bar是Activiti定义的一个扩展名，使用zip格式压缩）的压缩文件。

打包资源文件可以手动操作或使用Ant脚本，打包的文件扩展名可以是Activiti官方推荐的bar或普通的zip。

本节以candidateUserInUserTask和UserAnd-GroupInUserTask为例，目的是一次部署这两个流程的流程定义文件和对应的图片文件，文件列表参考图5-3。首先打包图5-3中的4个文件，命名压缩文件为“chapter5-deployment.bar”，如图5-4所示。

如果是手动打包文件，可以先用压缩工具把4个文件压缩成zip格式的文件，然后将其重命名为bar扩展名的文件，或者保持zip扩展名。

打包之后就可以通过Activiti的API部署打包的bar文件了，见代码清单5-13。

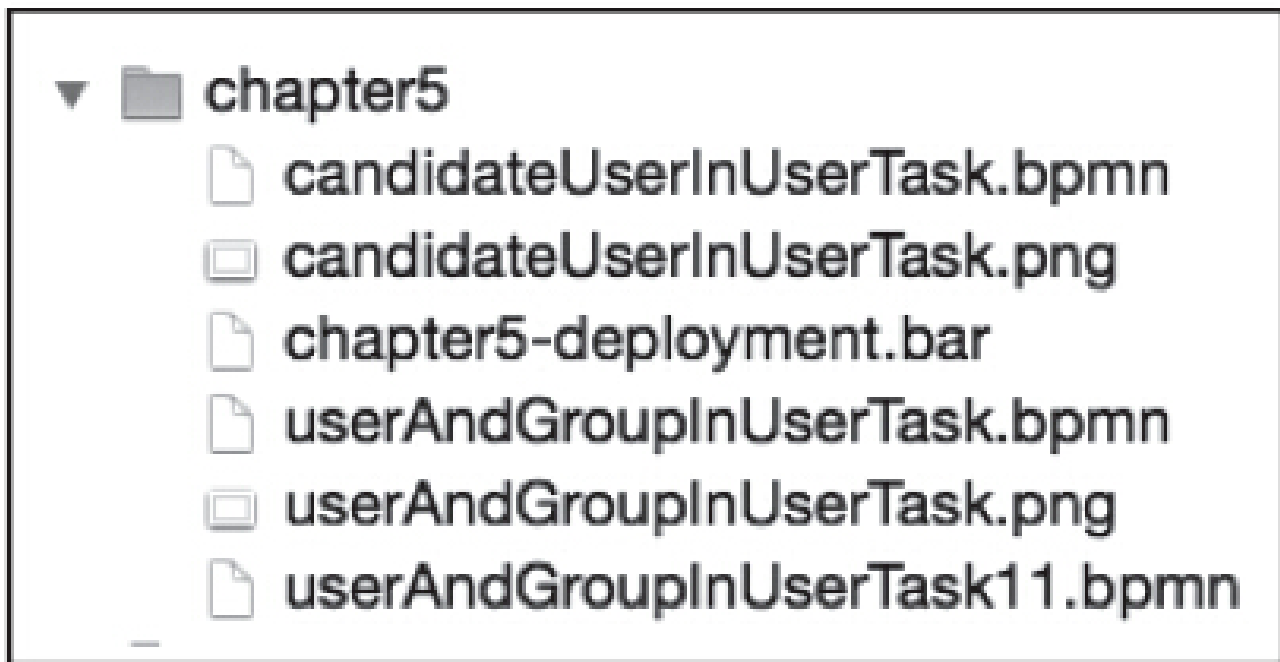


图5-4 打包4个资源文件并命名压缩文件为chapter5-deployment.bar

代码清单5-13 用压缩包方式部署流程资源文件

```
public class ZipStreamDeploymentTest extends AbstractTest {
    @Test
    public void testZipStreamFromAbsolutePath() throws Exception {
        // 从classpath读取资源并将其部署到引擎中
        InputStream zipStream = getClass().getClassLoader()
            .getResourceAsStream("chapter5/chapter5-deployment.bar"); #1
        repositoryService.createDeployment()
            .addZipInputStream(new ZipInputStream(zipStream)).deploy(); #2
        // 统计已部署流程定义的数量
        long count = repositoryService.createProcessDefinitionQuery().count();
        assertEquals(2, count); #3
        // 查询部署记录
        Deployment deployment = repositoryService.createDeploymentQuery().singleResult(); #4-S
        assertNotNull(deployment);
        String deploymentId = deployment.getId(); #4-E
        // 验证4个资源文件是否都已成功部署 #5-S
        assertNotNull(repositoryService.getResourceAsStream(deploymentId, "candidate UserInUserTask.bpmn"));
        assertNotNull(repositoryService.getResourceAsStream(deploymentId, "candidate UserInUserTask.png"));
        assertNotNull(repositoryService.getResourceAsStream(deploymentId, "userAnd GroupInUserTask.bpmn"));
        assertNotNull(repositoryService.getResourceAsStream(deploymentId, "userAnd GroupInUserTask.png")); #5-E
    }
}
```

代码清单5-13从classpath中读取一个bar文件部署，最后验证4个资源文件是否都部署到引擎中。下面——讲解关键代码的作用。

#1处代码的作用读者应该不陌生了，从classpath中读取一个资源并返回一个输入流对象。

#2处调用addZipInputStream()方法部署一个zip格式的压缩文件，引擎首先会将其解压缩，然后逐个部署单个资源文件，相当于用for循环方式执行了n次InputStream部署。

#3处验证已部署资源的数量为2。这里肯定有读者开始疑惑：压缩包中明明有4个文件，为什么部署之后只有两个？仔细看一下，代码#3处调用的方式是 repositoryService.createProcessDefinitionQuery()查询已部署的流程定义对象，在压缩包中，4个文件有两个扩展名为bpmn、两个扩展名为png，引擎会根据不同的扩展名进行不同的处理，bpmn（或 bpmn20.xml）类型的文件会在流程定义表中插入一条数据并在存放字节流表中插入数据，而PNG类型的文件则仅在存放字节流数据的表中插入一条数据，所以只有扩展名为bpmn的文件才会作为流程定义被查询到。

#4处用于查询一个部署记录对象，解释一下什么是部署：“一次部署可以包含单个或多个资源文件，调用一次DeploymentBuilder的deploy()方法即可称之为一次部署”；部署对象可以记录本次部署的时间，并且本次部署的资源文件记录中会关联本次部署的ID；在#4-E处的变量deploymentId即本次部署的一条记录的ID。

一图胜千语，大概了解几种部署资源的方式之后用一张图来说明执行的过程，如图5-5所示。

图5-5从“开始部署”处开始一次部署操作，然后保存本次所有的资源到字节流库中，紧接着处理（解析）本次部署的资源，流程引擎内置了两个处理器（解析流程定义、解析规则文件）用来解析常用的格式，部署完成后可能还有一些触发器需要执行，例如定时器启动事件要创建一个作业等。

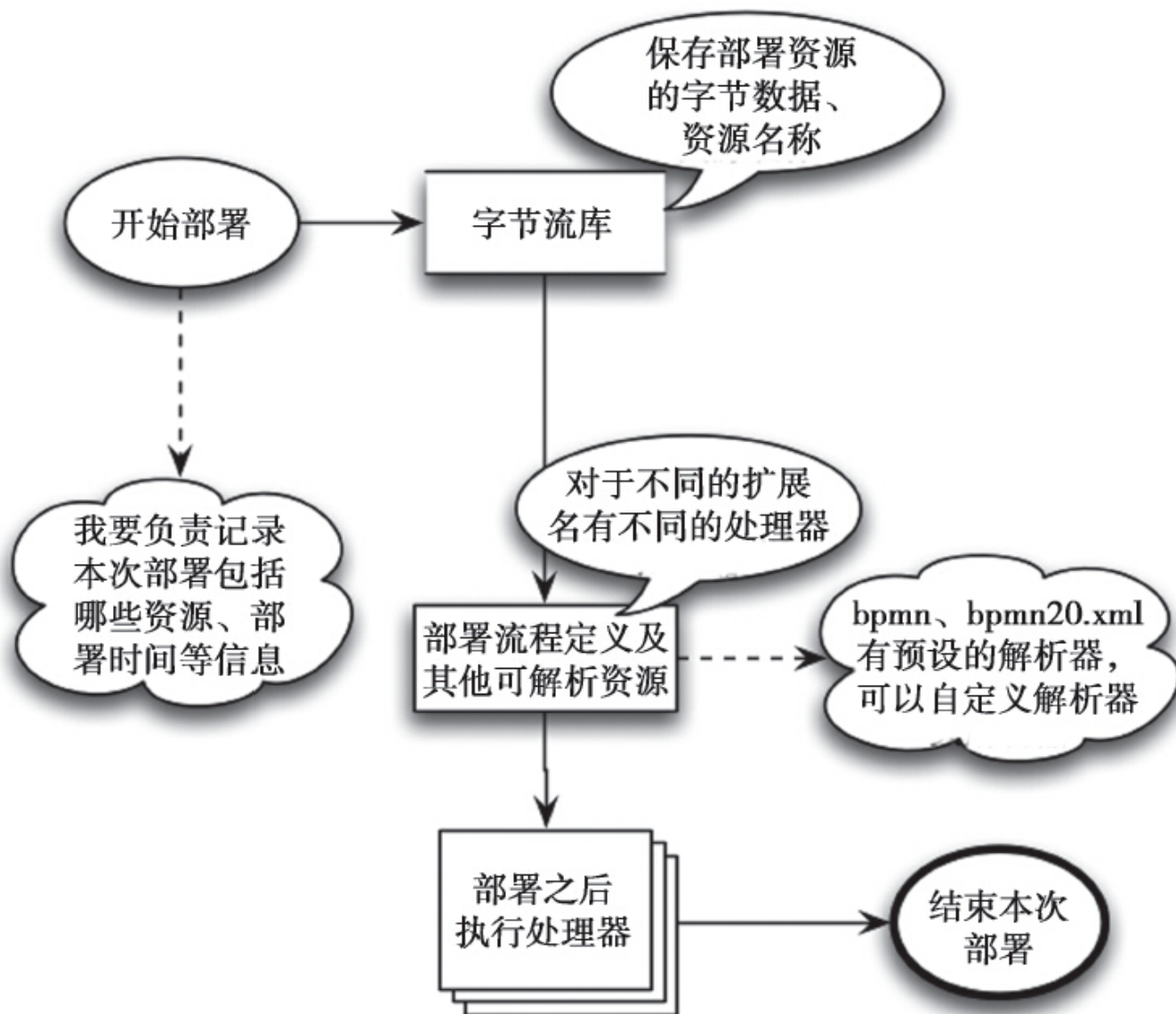


图5-5 一次部署执行的过程

5.3 流程部署及资源读取

在5.2节中学习了如何部署资源，本节将介绍如何利用API读取已部署的资源，例如读取流程定义的XML文件内容或流程对应的图片文件等。

本节又进一步接近企业应用中实际开发的需求，在Web的基础上构建一个管理流程部署的功能模块，可以查询已部署流程、上传文件部署流程、读取流程资源（XML、PNG）、删除已部署流程定义等。

本节的示例都是在目前最受欢迎的MVC框架Spring MVC的基础上实现的。Spring MVC是一个优秀的轻量级Web框架，和Spring无缝集成，采用松散耦合可插拔组件结构，相比其他的MVC框架有着更好的扩展性和灵活性。Spring MVC有一套相当成熟的注解支持，可以很容易让一个普通的POJO对象变成一个控制器（和Struts 2的Action功能相同）。本书中涉及Spring MVC的地方均使用注解方式定义Spring MVC的控制器（Controller）。

照顾不太熟悉Spring MVC的读者，先介绍几个注解的含义，示例代码见代码清单5-14。

代码清单5-14 快速理解Spring MVC的控制器的示例代码

```
@Controller
@RequestMapping(value = "/workflow")
public class ActivitiController {
    // 流程定义列表
    @RequestMapping(value = "/process-list")
    public ModelAndView processList() {
        ModelAndView mav = new ModelAndView("workflow/process-list");
        List<Object> objects = new ArrayList<Object>();
        mav.addObject("objects", objects);
        return mav;
    }
    // 部署全部流程
    @RequestMapping(value = "/redeploy/all")
    public String redeployAll() throws Exception {
        workflowProcessDefinitionService.deployAllFromClasspath();
        return "redirect:/workflow/process-list";
    }
}
```

· @Controller: 标记当前的类可以作为Spring MVC的控制器。

· @RequestMapping: 定义当前控制器的访问路径，路由由参数value指定，代码清单5-14中的控制器的访问路径就是/workflow。

· **ModelAndView**: Spring MVC中的模型和视图对象，顾名思义，可以在返回视图的同时设置模型数据变量，在返回的视图中就可以读取模型对象的变量值了。在创建**ModelAndView**对象时传入的参数即视图的路径，`mav.addObject("objects", objects)`把集合objects存放到mav中，`processList()`方法返回了mav对象，Spring MVC根据配置的视图响应客户端的请求。

· **redirect**: 重定向视图到冒号后面的视图路径。

本节内容的代码可以在chapter5-oa-manager项目的chapter5中找到。

5.3.1 读取已部署流程定义

流程定义在一个由流程驱动的业务系统中是必不可少的，所以第一步的任务就是要有一个列表可以浏览和管理流程定义，当然也要有对于流程定义相关的资源文件的浏览和管理。

图5-6展示的就是一个已部署流程定义列表。

流程定义ID	部署ID	流程定义名称	流程定义KEY	版本号	XML资源名称	图片资源名称	操作
--------	------	--------	---------	-----	---------	--------	----

图5-6 已部署流程定义列表

从图5-6的列表中可以读取已经部署到引擎数据库的流程定义，当然现在看到的是没有任何数据显示的空列表，暂且让它空着，重点是了解一下如何读取流程定义数据。

本节前面介绍了Spring MVC的一些特性以及几个关键的注解，现在看看图5-6中列表对应的Java代码是如何实现的，见代码清单5-15。

代码清单5-15 用Spring MVC实现读取已部署流程定义列表

```
@Controller
@RequestMapping(value = "/chapter5")
public class DeploymentController extends AbstractController {
    @RequestMapping(value = "/process-list") #1
    public ModelAndView processList() {
        ProcessEngine processEngine = ActivitiUtils.getProcessEngine(); #2
        RepositoryService repositoryService = processEngine.getRepositoryService();
        // 对应WEB-INF/views/chapter5/process-list.jsp
        ModelAndView mav = new ModelAndView("chapter5/process-list"); #3
        List<ProcessDefinition> processDefinitionList = #4
            repositoryService.createProcessDefinitionQuery().list();
        mav.addObject("processDefinitionList", processDefinitionList); #5
        return mav;
    }
}
```

代码清单5-15和代码清单5-14中讲解Spring MVC特性的例子类似，下面——说明重点代码的含义。

- #1处声明processList()方法的访问路径为process-list，该路径和类DeploymentController的@RequestMapping注解声明的value拼接起来就是：/chapter5/process-list。
- #2处用工具类获取Activiti引擎对象。
- #3处创建一个视图对象，参数“chapter5/process-list”对应的路径为“WEB-INF/views/chapter5/process-list.jsp”。
- #4处用于查询全部已经部署过的流程定义对象，也可以通过listPage()方法分页查询。
- #5处把查询结果设置到ModelAndView对象中，相当于调用request.setAttribute()方法。

本示例的View层使用JSP实现，对应的代码也比较简单，见代码清单5-16。

代码清单5-16 读取已部署流程定义的View代码

```
<table>
<thead>
    <tr>
        <th>流程定义ID</th>
        <th>部署ID</th>
        <th>流程定义名称</th>
        <th>流程定义KEY</th>
        <th>版本号</th>
        <th>XML资源名称</th>
        <th>图片资源名称</th>
    </tr>
</thead>
<c:forEach items="${processDefinitionList}" var="pd">
    <tr>
        <td>${pd.id}</td>
        <td>${pd.deploymentId}</td>
        <td>${pd.name}</td>
        <td>${pd.key}</td>
        <td>${pd.version}</td>
        <td>${pd.resourceName}</td>
        <td>${pd.diagramResourceName}</td>
    </tr>
</c:forEach>
</table>
```

代码清单5-16用<c:forEach>循环输出结果中的每一个流程定义对象，结果如图5-6所示。

5.3.2 从客户端部署流程

在上一小节中用Spring MVC和JSP实现了读取已部署流程定义的功能，但是读取的列表为空，本小节的任务就是从客户端部署流程定义，然后用上一小节的列表显示已部署流程定义及其资源。

在5.2节已经介绍了多种部署方式，在实际的企业应用中一般通过Web界面或其他客户端部署流程定义。本小节采用上传文件方式部署流程定义，支持压缩包方式和InputStream方式部署。

图5-7展示了部署流程资源的界面。

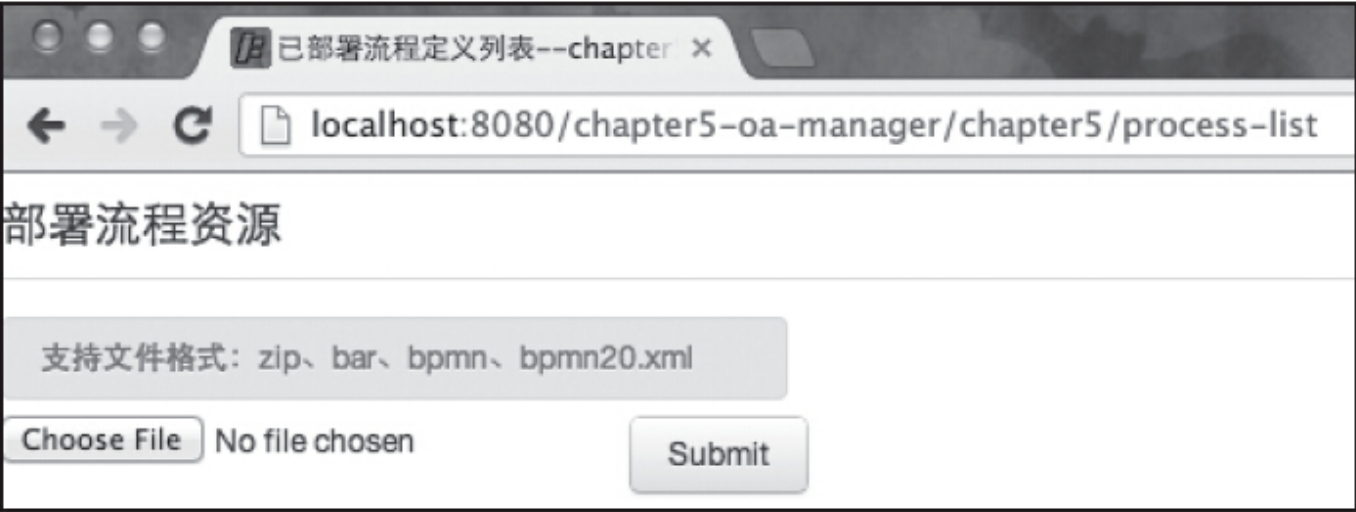


图5-7 部署流程资源的界面

图5-7对应的HTML代码见代码清单5-17。

代码清单5-17 图5-7对应的HTML代码

```
<form action="{ctx }/chapter5/deploy" method="post" enctype="multipart/form-data">
  <input type="file" name="file" />
  <input type="submit" value="Submit" class="btn" />
</form>
```

从Web端部署的过程比较简单，可以用图5-8来简单描述这一过程。

了解了部署过程之后再从代码的角度剖析过程细节，部署的代码还是在类Deployment-Controller中实现的，见代码清单5-18。

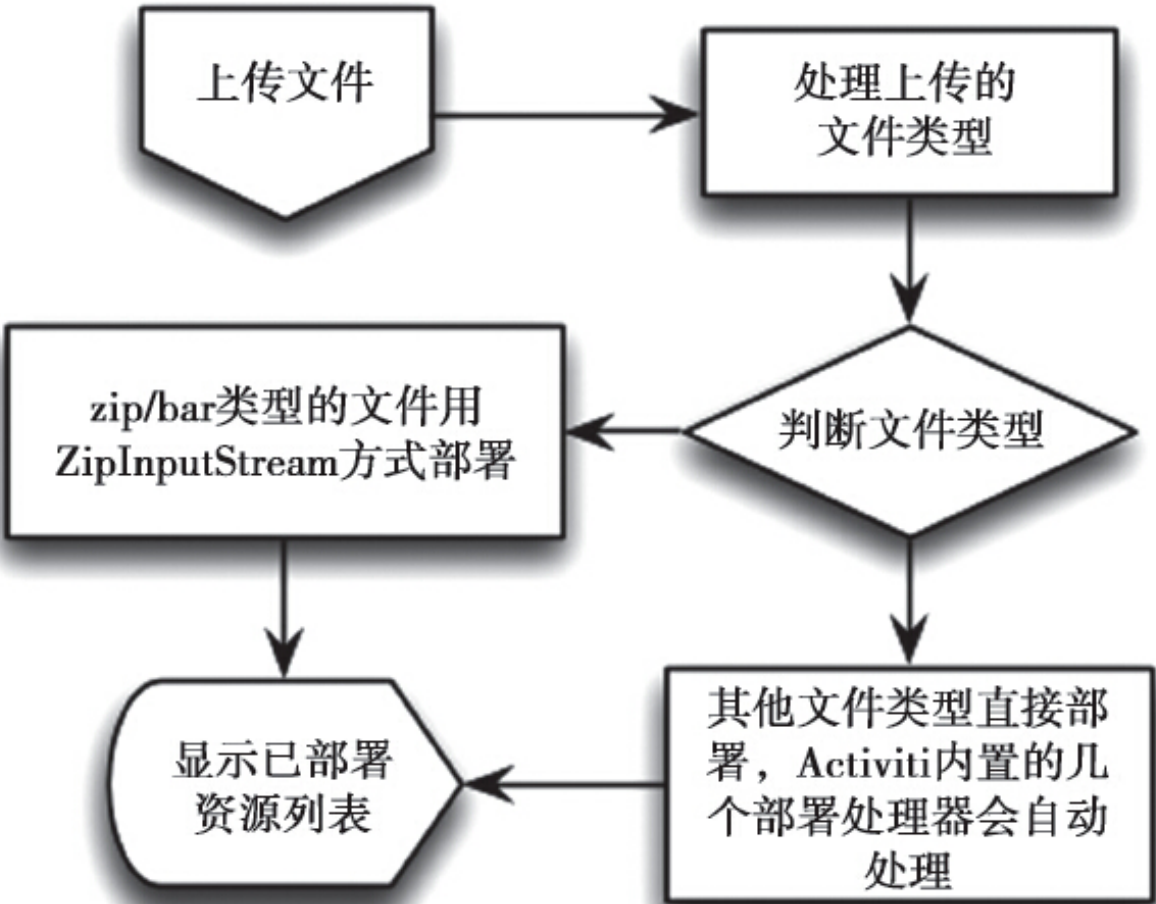


图5-8 从Web端部署流程资源的过程

代码清单5-18 部署流程资源的Java代码

```
@RequestMapping(value = "/deploy") #1
```

```
public String deploy(@RequestParam(value = "file") MultipartFile file) {      #2
RepositoryService repositoryService = processEngine.getRepositoryService();
// 获取上传的文件名
String fileName = file.getOriginalFilename();
try {
// 得到输入流（字节流）对象
InputStream fileInputStream = file.getInputStream();
// 文件的扩展名
String extension = FilenameUtils.getExtension(fileName);      #3
// zip或者bar类型的文件用ZipInputStream方式部署
DeploymentBuilder deployment = repositoryService.createDeployment();
if (extension.equals("zip") || extension.equals("bar")) {      #4-S
ZipInputStream zip = new ZipInputStream(fileInputStream);      #4-1
deployment.addZipInputStream(zip);      #4-E
} else {
// 其他类型的文件直接部署
deployment.addInputStream(fileName, fileInputStream);      #5
}
deployment.deploy();
} catch (Exception e) {
logger.error("error on deploy process, because of file input stream");
}
return "redirect:process-list"; #6
}
```

代码清单5-18中的代码正如图5-8所描述的过程那样，处理不同格式的资源并部署到引擎中。

#1处声明deploy()方法的访问路径为/deploy，即代码清单5-17中form的action属性。

#2处接受前端界面上上传的文件对象。

#3处用工具类获取文件的扩展名，对于不同类型的文件有不同的处理方式，所以要使用不同部署方式。

#4处判断类型为压缩文件后使用5.2.4节介绍的部署方式部署压缩包。

#5处不做任何判断直接部署除压缩包格式之外的资源，如果部署的文件类型不是压缩文件（如bpmn、bpmn20.xml或其他类型的文件），那么使用5.2.2节介绍的InputStream部署方式。

#6处部署后返回5.3.1节的流程定义列表。

在Web界面中选择了流程文件“candidateUserIn-UserTask.bpmn”，如图5-9所示。

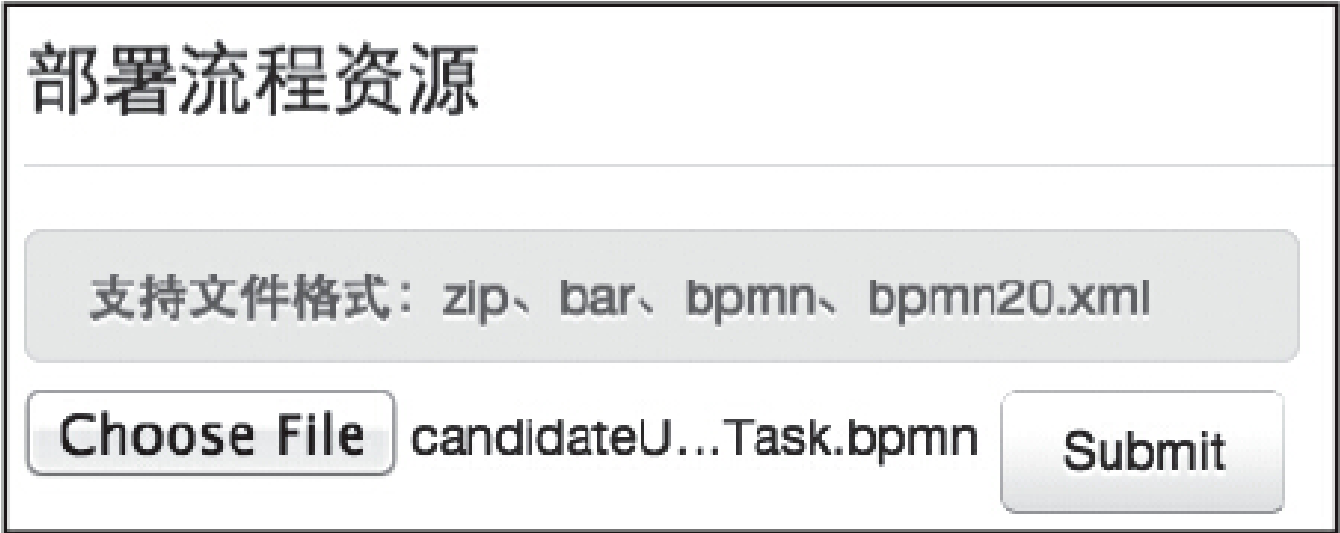


图5-9 在Web界面中选择部署的流程资源

在单击图5-9中的“Submit”按钮之后，页面刷新后显示刚刚部署的流程定义，如图5-10所示。

从图5-10中可以看出，列表中多出了一条数据，在5.3.3和5.3.4节将介绍如何读取已部署流程定义的XML和图片文件。

已部署流程定义列表--chapter x

localhost:8080/chapter5--oa-manager/chapter5/process-list

部署流程资源

支持文件格式: zip、bar、bpmn、bpmn20.xml

Choose File No file chosen Submit

流程定义ID	部署ID	流程定义名称	流程定义KEY	版本号	XML资源名称	图片资源名称	操作
candidateUserInUserTask:1:4	1	candidateUserInUserTask	candidateUserInUserTask	1	candidateUserInUserTask.bpmn	candidateUserInUserTask.candidateUserInUserTask.png	删除

图5-10 部署流程后列表显示了部署成功的流程定义

在代码清单5-13的#4处已经涉及了读取资源的代码，对应的代码为：repository-Service.getResourceAsStream(deploymentId, "candidateUserInUserTask.bpmn")。

本小节在图5-10的基础上为列表中的“XML资源名称”添加一个链接，用来在单击后查看流程定义的XML文件内容，把原来的“<td>\${pd.resourceName}</td>”替换为下面的代码。

```
"<td>${pd.resourceName }</td>"替换为下面的代码。
<td><a target=" _blank"
href='${ctx }/chapter5/Read-resource?pdid=${pd.id }&resourceName=${pd.resourceName }'>${pd.resourceName }</a></td>
```

URL中的“pdid”参数为流程定义ID，“resourceName”为资源的名称，在列表的“XML资源名称”单元格中就显示为一个链接，如图5-11所示。



图5-11 为“XML资源名称”添加链接后的显示效果

有了链接后的任务就是用代码实现读取XML内容的功能，不过在讲解代码之前要说明一下：所有的流程资源（不管是XML、PNG格式还是其他格式的资源）均通过一个方法“repositoryService.getResourceAsStream()”读取。代码清单5-19列出了一个通用的读取资源内容的方法。

代码清单5-19 读取流程资源的Java代码

```
@RequestMapping(value = "/read-resource")
public void readResource(@RequestParam("pdid") String processDefinitionId, @RequestParam("resourceName") String resourceName, HttpServletResponse response)
    throws Exception {
    ProcessDefinitionQuery pdq = repositoryService.createProcessDefinitionQuery();
    ProcessDefinition pd = pdq.processDefinitionId(processDefinitionId).singleResult();
    // 通过接口读取资源流
    InputStream resourceAsStream = repositoryService #1-S
        .getResourceAsStream(pd.getDeploymentId(), resourceName); #1-E
    // 输出资源内容到相应对象
    byte[] b = new byte[1024]; #2-S
    int len = -1;
    while ((len = resourceAsStream.read(b, 0, 1024)) != -1) {
        response.getOutputStream().write(b, 0, len);
    } #2-E
}
```

在代码清单5-19的#1处通过部署ID和资源名称得到了一个字节流对象，在#2处把资源的字节流输出到HttpServletResponse中。单击图5-11的链接之后可以看到如图5-12所示的结果。



图5-12 通过单击链接显示XML资源文件内容

5.3.4 读取流程定义的图片及图片中的中文乱码

流程定义的图片资源是和流程定义关联在一起的，它的基本作用是可以让用户清楚整个流程需要处理的任务及流程的走向，还有一个作用是在流程运行过程中显示地跟踪流程处理的进度，这一点在第6章中将会详细介绍。

一个流程定义会对应一个图片资源。图片可以由引擎自动生成，或者与流程定义一起部署（压缩包方式），这样引擎不再自动生成图片资源，而是使用部署包中的图片资源。图5-13很好地描述了这一过程。

从图5-13中可以看出，是否生成图片分为两种情况：如果仅仅部署了扩展名为bpmn或bpmn20.xml的文件会自动生成图片；如果部署的是一个压缩包，这个压缩包中可能包含很多个文件，甚至多个

流程定义文件，若压缩包中不包含和流程定义同名的图片文件，则引擎同样会自动生成图片文件。

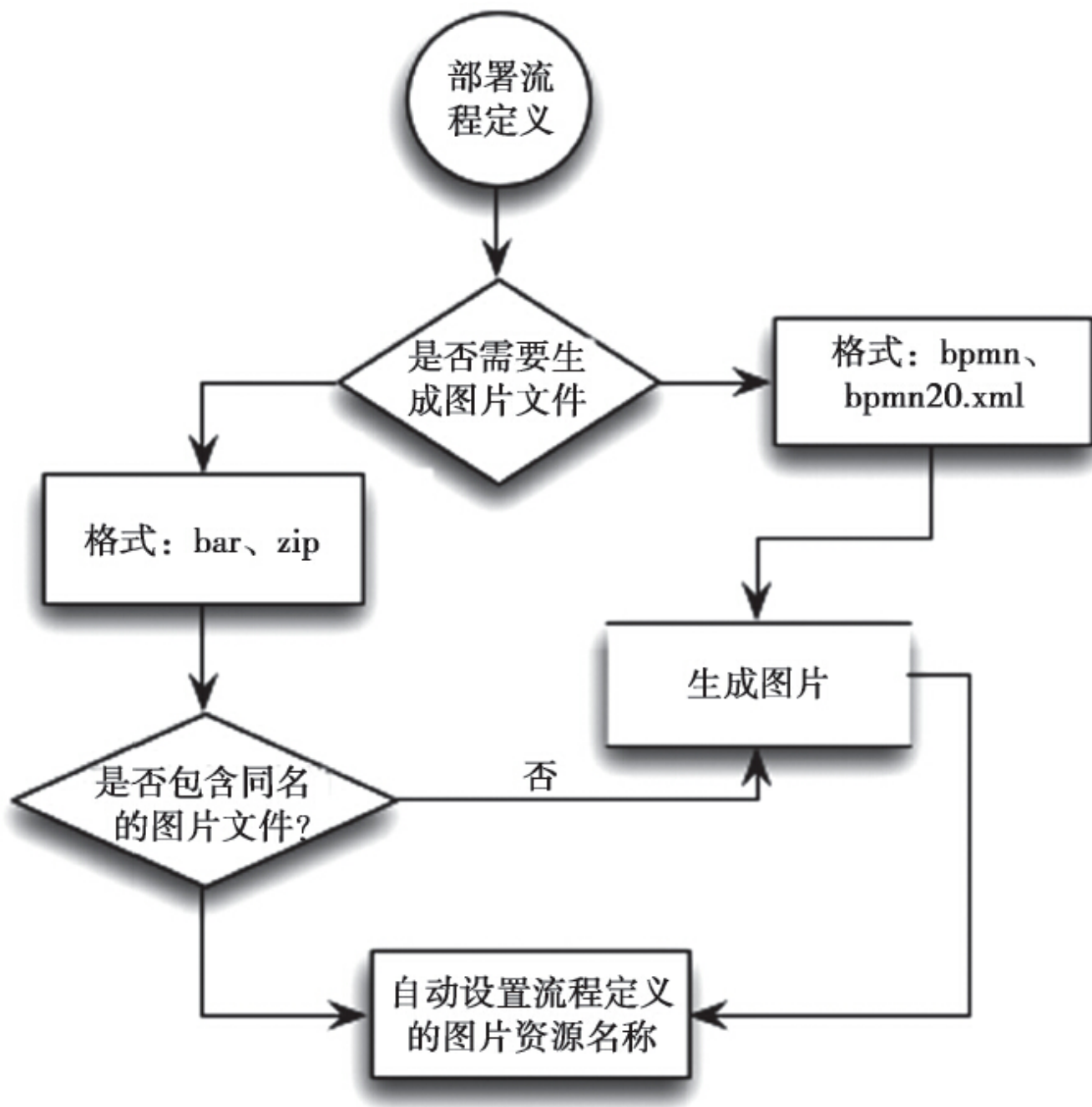


图5-13 流程引擎生成流程图片的过程

首先通过示例方式展示引擎自动生成的图片资源。在上一小节中学习了如何读取流程定义的XML资源内容，知道了不管XML还是PNG都统称为“资源”，而且也了解到读取所有资源都通过一个接口完成，所以读取图片资源和读取XML资源的方式一样。

将代码清单5-16中显示图片资源名称的单元格用如下代码替换，单元格内容是一个超级链接，显示效果如图5-14所示。

```
<td><a target="" blank"
href='${ctx }/chapter5/read-resource?pdid=${pd.id }&resourceName=${pd.diagram-ResourceName }'>${pd.diagramResourceName }</a></td>
```



图5-14 添加了超级链接的图片资源名称的单元格显示效果

单击图5-14中的链接之后在新窗口中显示了引擎自动生成的图片文件，如图5-15所示。

在部署流程定义文件时只部署了一个bpmn（或bpmn20.xml）为扩展名的文件，根据图5-13的流程可以判定图5-15的流程图是引擎自动生成的。除了这种情况外还有一种情况会自动生成图片文件，那就是部署一个压缩包文件，但是其中仅仅包含流程定义文件。



图5-15 在打开的新窗口出现了引擎自动生成的图片

还是以流程candidateUserInUserTask为例，图5-16是用压缩工具生成的一个压缩文件，只包含“candidateUserInUserTask.bpmn”一个文件。



图5-16 只包含一个bpmn文件的压缩文件

完成部署之后在已部署流程定义列表中多出一条版本号为2的记录，如图5-17所示。

流程定义ID	部署ID	流程定义名称	流程定义KEY	版本号	XML资源名称	图片资源名称
candidateUserInUserTask:1:4	1	candidateUserInUserTask	candidateUserInUserTask	1	candidateUserInUserTask.bpmn	candidateUserInUserTask.candidateUserInUserTask.png
candidateUserInUserTask:2:8	5	candidateUserInUserTask	candidateUserInUserTask	2	candidateUserInUserTask.bpmn	candidateUserInUserTask.candidateUserInUserTask.png

图5-17 在已部署流程定义列表多出了一条记录

通过单击版本号为2的记录的两个链接可以看出，结果和单击版本号为1的记录一样，生成的图片和图5-15也是一样，这也再次证明了图5-13中的生成图片的逻辑。看到了图片是不是突然觉得学习Activiti的动力十足了呢？但是这里不得不打击一下大家，有一个老生常谈的问题，那就是“中文乱码”。图5-18展示了流程图中的中文乱码。导致乱码的原因是字体库的原因，因为Activiti引擎默认字体名称为“Arial”，而“Arial”不支持中文字符，解决办法也不难，更换一个支持中文字符的字体即可。

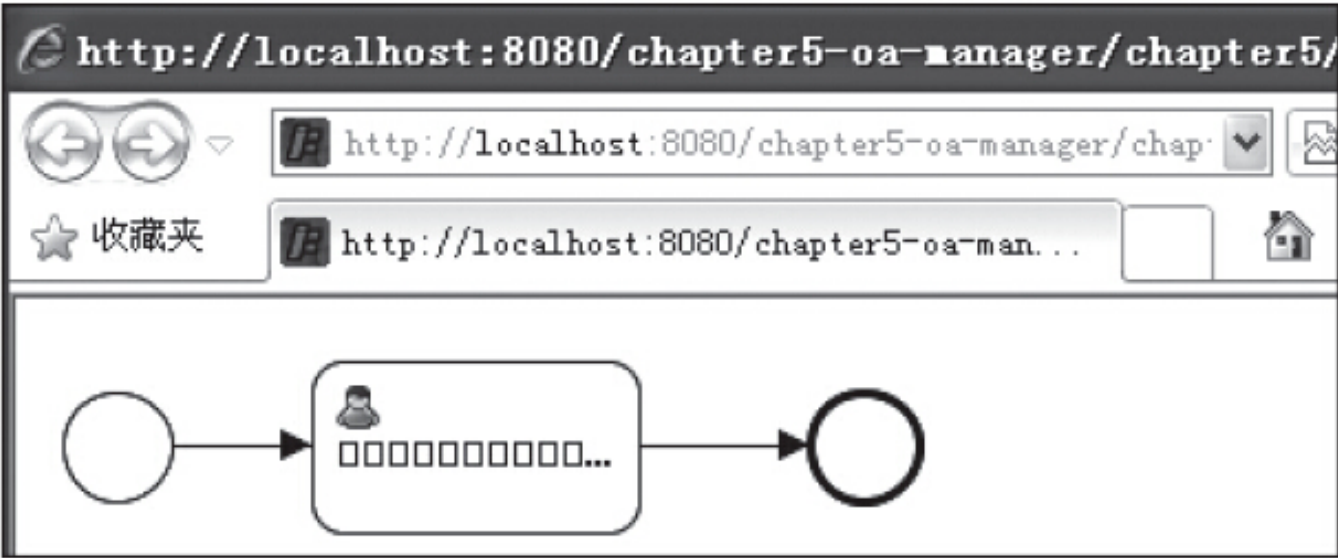


图5-18 引擎自动生成的图片中的中文乱码

说明 读者比较疑惑的可能是：为什么图5-15在没有更改字体之前没有出现乱码问题？这和操作系统有关系，笔者使用的是Mac OS X系统，而图5-18的乱的问题是在Windows平台上产生的，所以在Windows平台上使用引擎自动生成图片需要更改字体名称。

为了解决中文字符的乱码问题不得不侵入源码内部进行更改，以便解决这个读者遇到最多的问题。Activiti生成图片使用的是Awt（JDK中提供的画图组件），负责生成图片的Java类为“org.activiti.engine.impl.bpmn.diagram.ProcessDiagramCanvas”，此类中有一行关键代码“Font font = new Font (“Arial”, Font.BOLD, 11);”，其中加粗部分的“Arial”即引擎默认的字体文件。中文版的Windows都会默认自带“宋体”，Vista以上版本自带“微软雅黑”等中文字体，要解决流程图中的中文乱码问题，只需要把默认字体文件改为操作系统中支持中文的字体文件的名称即可。图5-19显示了把“Arial”改为宋体文件的名称“simsum”之后的效果，对应的代码如下：



图5-19 改用“宋体”字符之后生成的图片正常显示中文字符

流程引擎自动生成的图片虽然解决了乱码问题，但是解决的手段比较“另类”，需要入侵修改源代码，这会导致在升级版本时需要同步类“org.activiti.engine.impl.bpmn.diagram.ProcessDiagramCanvas”，并且再次修改字体名称。细心的读者可能已经发现了，当任务中的文字比较长时会用“...”代替，这在实际应用中是绝对不允许出现的，在未来版本中会修复此问题（可以通过修改源码的方法解决）。

在国内的Activiti社区中经常有人问到流程图中文乱码如何解决，除了刚刚介绍的修改引擎源码中字体的配置方法之外，笔者在Activiti 5.12版本发布之前对此做了修复，允许通过配置的方式更改字体名称，例如，在如下的配置中把字体更改为了“宋体”。

```
<bean id="processEngineConfiguration"
      class="org.activiti.xxx.XxxProcessEngineConfiguration">
  http://www.hzcourse.com/resource/readBook?path=/openresources/teach_ebook/uncompressed/15030/OEBPS/Text/...
  <!-- 生成流程图的字体 -->
  <property name="activityFontName" value="宋体"/>
  <property name="labelFontName" value="宋体"/>
</bean>
```

修改字体名称的解决办法不完美，是不是还有另外的办法解决中文乱码问题呢？答案是肯定的。简单来说就是把流程定义文件和同名的流程图片文件一起打包后部署。暂时不理解没关系，下面结和图来进行解释。图5-20展示了一个名称为“userAndGroupIn-UserTask-with-bpmn-and-png.zip”的压缩文件，它包含了两个文件，分别为“userAnd-GroupInUserTask.bpmn”和“userAndGroupInUserTask.png”。

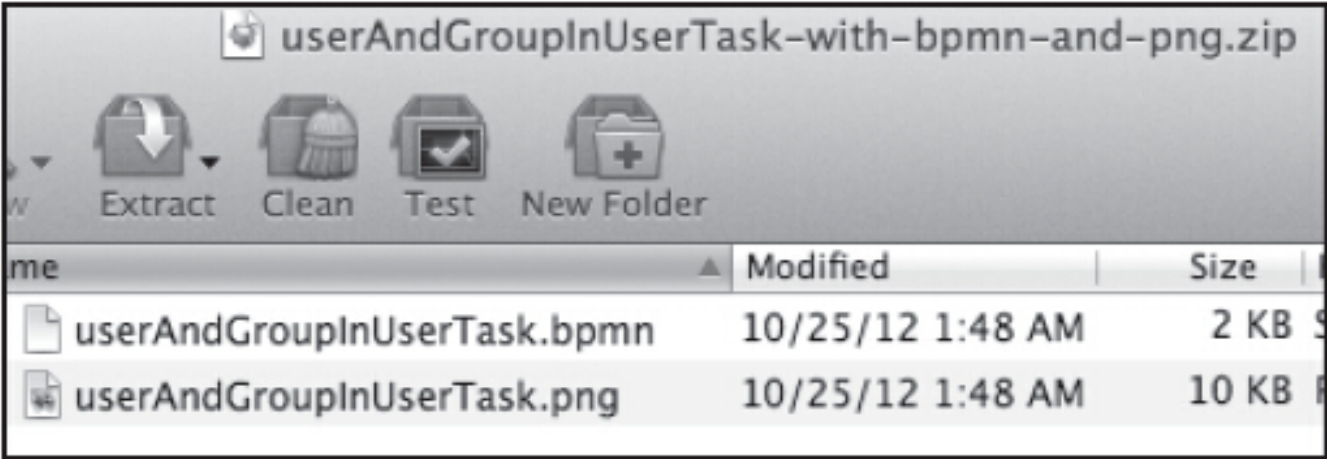


图5-20 压缩文件userAndGroupInUserTask-with-bpmn-and-png.zip

接着部署“userAndGroupInUserTask-with-bpmn-and-png.zip”文件，在已部署列表中显示了一个刚刚部署的userAndGroupInUserTask流程定义，如图5-21所示。

流程定义ID	部署ID	流程定义名称	流程定义KEY	版本号	XML资源名称	图片资源名称
candidateUserInUserTask:1:4	1	candidateUserInUserTask	candidateUserInUserTask	1	candidateUserInUserTask.bpmn	candidateUserInUserTask.candidateUserInUserTask.png
userAndGroupInUserTask:1:8	5	userAndGroupInUserTask	userAndGroupInUserTask	1	userAndGroupInUserTask.bpmn	userAndGroupInUserTask.png

图5-21 通过压缩文件userAndGroupInUserTask-with-bpmn-and-png.zip部署的流程定义

注意 图5-21中的第一条记录是通过部署candidateUserInUserTask.bpmn文件生成的，第二条记录是通过部署userAndGroupInUserTask-with-bpmn-and-png.zip文件生成的。不同的方式也导致“图片资源名称”显示的不同：由引擎自动生成的图片文件名称为“candidateUserInUserTask.candidateUserInUserTask.png”，即两个用“.”分割的流程定义的Key组成；通过压缩包方式（包含图片）部署的流程定义显

示的“图片资源名称”保持不变，如果压缩包中包含多级目录，则路径部分用“/”分割，例如，“aaa/bbb/userAndGroupInUserTask.png”。

单击图5-21中的第二条流程定义记录的“图片资源名称”的超级链接，可以看到和图5-1相同的图片，如图5-22所示。

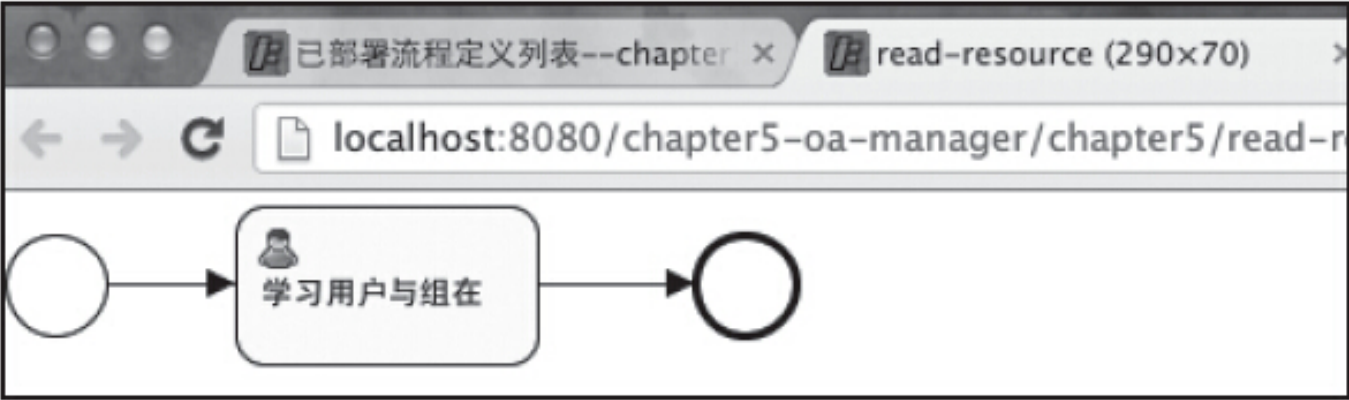


图5-22 通过压缩包方式（包含流程定义图片）部署之后查看图片资源

从图5-18和图5-22中可以比较出另外一个不同之处，图片中元素的坐标是不同的。图5-18是靠顶部开始渲染图片的，原因是引擎在生成图片的时候丢失了坐标信息，而图5-22和设计时保持一致。

两种解决中文乱码的方式各不相同，表5-1列出了两种方式的区别。

表5-1 两种解决中文乱码方式的区别

特 性	引擎自动生成	压缩包方式部署
图片元素的完整性	过长的文字被截取、坐标丢失 (虽然都可以通过修改源码解决)	和部署时的图片一致
灵活性（添加其他附加文字，美化图片等）	不支持	和部署时的图片一致
部署方式	仅需提供流程定义文件	必须提供流程定义文件和同名的图片文件，并且要打包为压缩文件

除了使用引擎的AWT提供的流程图生成功能外，还可以使用Activiti官方提供的Diagram Viewer组件生成流程图，该组件使用前端的Javascript、SVG（不支持SVG时使用VML）、CSS等技术实现，不过需要Activiti Rest模块提供服务。有关该组件的集成与使用在20.6节有详细介绍。

5.3.5 删除部署

前面花了不少篇幅讲解如何部署和读取流程资源，本小节将讲解如何删除已经部署的流程定义（包括图片资源）。

我们以流程管理用例为出发点，首先在如图5-10所示的流程定义列表页面添加一个删除部署的入口，如图5-23所示。

流程定义ID	部署ID	流程定义名称	流程定义KEY	版本号	XML资源名称	图片资源名称	操作
userAndGroupInUserTask:1:8	5	userAndGroupInUserTask	userAndGroupInUserTask	1	userAndGroupInUserTask.bpmn	userAndGroupInUserTask.png	删除

5为部署的ID

删除操作的入口

图5-23 在列表中添加删除部署的超级链接

一个流程定义不能直接删除，而需要通过流程定义的部署ID删除。在执行删除操作时会同时删除所有和本次部署有关的资源，如果只部署了一个bpmn或bpmn20.xml流程定义文件，则会删除流程定义及引擎生成的图片资源；如果使用压缩包来部署资源，则将其全部删除。

图5-23中“删除”对应的代码如下：

```
<td><a target="_blank"
href='${ctx }/chapter5/delete-deployment?deploymentId=${pd.deploymentId }'>删除</a></td>
```

删除部署的Java代码见代码清单5-20。

代码清单5-20 删除部署的Java代码

```
@RequestMapping(value = "/delete-deployment")
public String deleteProcessDefinition(@RequestParam("deploymentId") String deploymentId) {
    // 调用删除部署的API
    repositoryService.deleteDeployment(deploymentId, true); #1
    return "redirect:process-list";
}
```

代码清单5-20比较简单，除了接受参数返回列表之外，最重要的就是#1处调用RepositoryService接口的deleteDeployment()方法，通过传入一个部署的ID删除一个流程；第二个参数“true”表示该

部署包含的流程定义已经被运行中或已结束的流程实例使用，因此同时把和流程相关的流程数据也一并删除。

5.4 本章小结

本章介绍了Activiti流程引擎的两个比较重要的“服务性”功能：用户与组、部署管理，对每一个功能最重要的理论及API的使用都进行了详细说明，并且通过实战方式进一步加深理解。

在关于用户与组方面，首先单独介绍每一个对象的管理方法，然后讲解了如何结合流程实例中的用户任务一起使用，最后说明了候选组与候选人之间的区别及关系。

在关于流程部署方面，用单独的篇幅介绍了多种部署方式，详细地说明了每种部署方式的实现方法及应用场合，在5.2.4节用流程图的方式解释了一个资源部署到引擎的过程。

在5.3节中介绍了如何读取已部署资源，包括列表、XML文件、图片文件等，还有很多读者最关心的中文乱码问题的解决方案。

本章内容作为“实战篇”的基础组件存在，熟悉本章内容对于掌握本篇其他章节内容大有帮助，涉及用户与组及流程部署方面的功能都依赖本章内容的功能实现。

第6章 任务表单

本章将一步一步完成一个协同办公系统（OA）的请假流程的设计，从实用的角度来考虑问题，讲解如何把Activiti和实际的业务紧密结合起来。

在4.1.1节中简单描述了两类表单的区别，即动态表单和外置（FormKey）表单的区别，本章将详细介绍如何使用不同的表单完成相同的功能。

6.1 动态表单

在第2、3、4章中已经涉及了请假流程的设计，但是业务逻辑的描述都比较粗糙，本章的请假流程作为一个接近于实际应用，所以需要重新设计。重新设计的请假流程定义如图6-1所示。

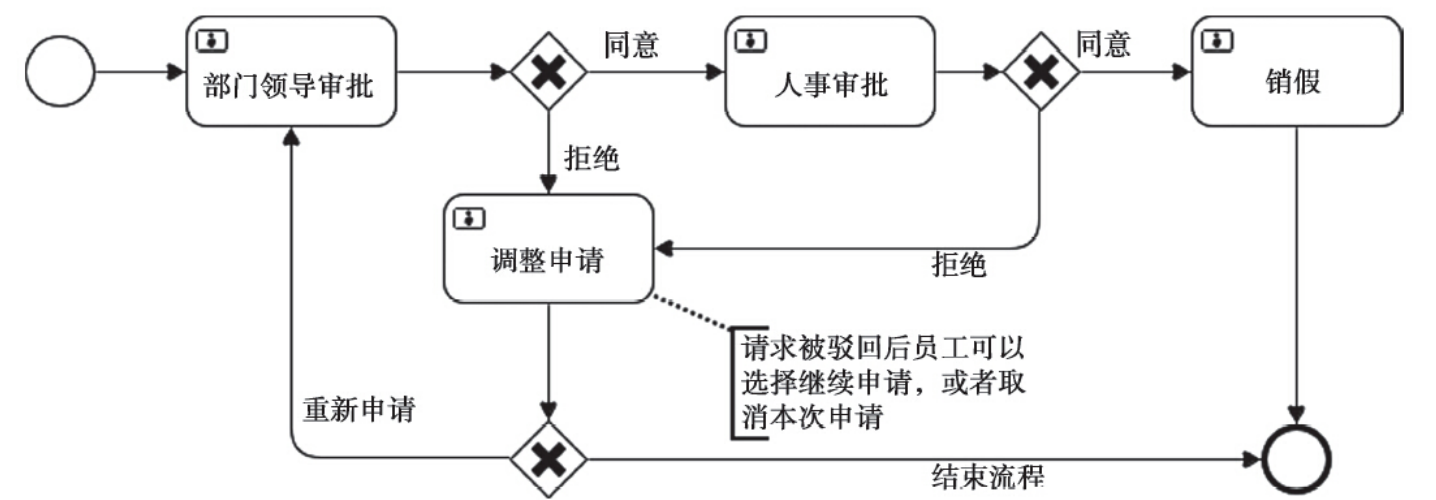


图6-1 请假流程的业务流程图

图6-1描述了请假流程的几个用户任务及业务逻辑的判断，首先员工自主启动一个请假流程，然后由部门经理和人事经理审批，全部审批通过之后流程流转至销假节点，申请人完成“销假”任务后结束本次流程。如果部门经理或人事经理审批没有通过，则由员工自己选择是调整信息后继续申请还是取消本次申请。

 **提示** 本节内容均存放在本书配套资源文件的“chapter6-oa-forms”项目中。

6.1.1 流程定义

由于流程的代码比较多，图6-1的XML代码将以分段方式讲解，先来看代码清单6-1。

代码清单6-1 图6-1请假流程的开始节点XML定义

```
<process id="leave" name="请假流程-动态表单">
<startEvent id="startevent1" name="Start" activiti:initiator="applyUserId">
<extensionElements>
<activiti:formProperty id="startDate" name="请假开始日期" type="date"
datePattern="yyyy-MM-dd" required="true" readable="true"
writable="true"></activiti:formProperty>
<activiti:formProperty id="endDate" name="请假结束日期" type="date"
datePattern="yyyy-MM-dd" required="true" readable="true"
writable="true"></activiti:formProperty>
<activiti:formProperty id="reason" name="请假原因" type="string" required="true"
readable="true" writable="true"></activiti:formProperty>
</extensionElements>
</startEvent>
</process>
```

在4.1.1节中已经简单说明了activiti:initiator属性的作用：可以把启动流程实例的操作人以变量名称“applyUserId”保存到数据库中，需要配合identityService.setAuthenticated-UserId (String userId) 方法使用，其中userId即当前操作人，在实际的应用中应该是当前的用户ID，引擎会把setAuthenticatedUserId()方法的参数作为流程启动人，通过调用HistoricProcessInstance实例的getStartUserId()可以获取一个历史（也可能正在运行）流程实例由哪个用户启动。要获取设置的用户ID，可以通过调用Authentication.getAuthenticatedUserId()方法来实现。

从代码清单6-1中可以看出空启动事件一共定义了三个字段，分别用来描述请假的开始日期、结束日期和请假原因。每一个字段都配置了几个属性，每个属性的含义如表6-1所示。

表6-1 activiti:formProperty属性列表

属性名称	属性说明
id	字段的唯一标识，对应 html 元素的 id 属性，在保存字段时就是用此字段作为 Key
name	字段的 Name，用来描述字段的名称
type	字段的类型，Activiti 默认支持几种类型： ☐ string，对应类为 org.activiti.engine.impl.form.StringFormType ☐ long，对应类为 org.activiti.engine.impl.form.LongFormType ☐ enum，对应类为 org.activiti.engine.impl.form.EnumFormType ☐ date，对应类为 org.activiti.engine.impl.form.DateFormType ☐ boolean，对应类为 org.activiti.engine.impl.form.BooleanFormType 除了以上的几种类型，也可以自定义字段类型以满足实际需求
value	字段的值
expression	表达式，可以通过计算表达式设置字段的值，例如 expression= " #{leave.reason}" 可以从变量名为 leave 的变量中获取属性 reason 的值
variable	将字段的值以 variable 指定的变量名保存
default	字段的默认值
datePattern	属性 type 的值为“date”时需要设置此属性定义日期格式，例如，yyyy-MM-dd 表示 2012-12-20
readable	是否可以读
writable	是否可写
readable	是否必填项
form values	这个不是属性，而是当 type= "enum " 时需要指定多个可选值，在 activiti:formKey 标签中嵌入多个 activiti:value 标签，例如： <pre><activiti:formProperty id="deptLeaderPass" name=" 审批意见" type="enum" required="true" writable="true"> <activiti:value id="true" name=" 同意 "/> <activiti:value id="false" name=" 拒绝 "/> </activiti:formProperty></pre>

表6-1除了列出的字段属性之外，还有一个针对于type="enum"的特殊标签<activiti:formProperty/>，可以在标签内定义多个<activiti:value/>来表示多个候选值，适用于页面中的下拉框（html的select组件），典型的配置可以参考代码清单6-2的#1-1~#1-2处。

代码清单6-2列出了用户任务“部门领导审批”的XML定义。

代码清单6-2 图6-1请假流程的部门领导审批用户任务的XML定义

```
<userTask id="deptLeaderAudit" name="部门领导审批" #1-S
activiti:candidateGroups="deptLeader"> // 把任务分配给角色"deptLeader"
<extensionElements>
<activiti:formProperty id="startDate" name="请假开始日期" type="date"
value="${startDate}" datePattern="yyyy-MM-dd" readable="true"
writable="false"></activiti:formProperty>
<activiti:formProperty id="endDate" name="请假结束日期" type="date"
value="${endDate}" datePattern="yyyy-MM-dd" readable="true"
writable="false"></activiti:formProperty>
<activiti:formProperty id="reason" name="请假原因" type="string" value="${reason}"
readable="true" writable="false"></activiti:formProperty>
<activiti:formProperty id="deptLeaderApprove" name="审批意见" type="enum" #1-1
required="true" writable="true">
<activiti:value id="true" name="同意"></activiti:value>
<activiti:value id="false" name="拒绝"></activiti:value>
</activiti:formProperty> #1-2
</extensionElements>
</userTask> #1-E
<exclusiveGateway id="exclusiveGateway5" name="Exclusive Gateway"/> #2
<sequenceFlow id="flow4" name="拒绝"
sourceRef="exclusiveGateway5" targetRef="modifyApply">
<conditionExpression xsi:type="tFormalExpression">
<![CDATA[${deptLeaderApprove== 'false'}]]></conditionExpression>
</sequenceFlow>
<sequenceFlow id="flow5" name="同意" sourceRef="exclusiveGateway5" targetRef="hrAudit">
<conditionExpression xsi:type="tFormalExpression">
<![CDATA[${deptLeaderApprove== 'true'}]]></conditionExpression>
</sequenceFlow>
```

从代码清单6-2的#1-S处开始定义了用户任务的表单，该表单多了一个“审批意见”字段，供部门领导决定本次申请的处理结果，是同意申请还是拒绝申请，设置了activiti:formProperty的类型为“enum”并提供了两个候选值作为SELECT元素的选项。

从代码清单6-2的#2处开始定义了用户任务“部门领导审批”在不同条件下处理方式：当表达式“`${deptLeaderApprove}==false`”成立时，将输出流指定到ID为“`modifyApply`”的用户任务，即“调整申请”；当表达式“`${deptLeaderApprove}==true`”成立时，将输出流指定到ID为“`hrAudit`”的用户任务，即“人事审批”。

代码清单6-3列出了用户任务“人事审批”的XML定义，结构和“部门领导审批”类似，不同的仅仅是审批意见的字段ID不同。

代码清单6-3 图6-1请假流程的人事审批用户任务的XML定义

```
<userTask id="deptLeaderAudit" name="人事审批" activiti:candidateGroups="hr"> #1
  <extensionElements> // 把任务分配给角色"hr"
    ..... // 省略了请假日期和请假原因字段定义
  <activiti:formProperty id="hrApprove" name="审批意见" type="enum"
    required="true" writable="true">
    <activiti:value id="true" name="同意"></activiti:value>
    <activiti:value id="false" name="拒绝"></activiti:value>
  </activiti:formProperty>
</extensionElements>
</userTask>
<exclusiveGateway id="exclusivegateway6" name="Exclusive Gateway" /> #2
<sequenceFlow id="flow6" name="" sourceRef="hrAudit" targetRef="exclusivegateway6"/>
<sequenceFlow id="flow7" name="同意" sourceRef="exclusivegateway6"
  targetRef="reportBack">
  <conditionExpression xsi:type="tFormalExpression">
    <![CDATA[${hrApprove == 'true'}]]></conditionExpression>
  </sequenceFlow>
<sequenceFlow id="flow9" name="拒绝" sourceRef="exclusivegateway6"
  targetRef="modifyApply"> //
  <conditionExpression xsi:type="tFormalExpression">
    <![CDATA[${hrApprove == 'false'}]]></conditionExpression>
  </sequenceFlow>
```

值得注意的是用户任务“部门领导审批”和“人事审批”请假相关字段的“`writable`”属性都为“`false`”，只有“审批意见”字段的“`writable`”属性为“`true`”。

从代码清单6-3的#2处开始定义了用户任务“人事审批”在不同条件下的处理方式：当表达式“`${hrApprove}==false`”成立时，将输出流指定到ID为“`modifyApply`”的用户任务，即“调整申请”；当表达式“`${hrApprove}==true`”成立时，将输出流指定到ID为“`reportBack`”的用户任务，即“销假”。

假如部门领导和人事经理都审批通过了，那么现在任务就会再次交由申请人办理，也就是“销假”节点，由申请人填写实际的请假时长。销假用户任务的XML描述如代码清单6-4所示。

代码清单6-4 图6-1请假流程的销假用户任务的XML定义

```
<userTask id="reportBack" name="销假" activiti:assignee="${applyUserId}"> #1
  <extensionElements>
    ..... // 请假的信息和部门领导审批节点相同
  <activiti:formProperty id="reportBackDate" name="销假日期" type="date" #2
    datePattern="yyyy-MM-dd" required="true" readable="true"
    default="${endDate}" writable="true"></activiti:formProperty>
  </extensionElements>
</userTask>
<sequenceFlow id="flow8" name="销假" sourceRef="reportBack" targetRef="endevent1">
  <extensionElements>
    <activiti:executionListener event="take"
      expression="${execution.setVariable('result', 'ok')}" /> #3
  </extensionElements>
</sequenceFlow>
```

在代码清单6-4中需要读者特别注意，#1处的`activiti:assignee`属性使用表达式的方式指定任务办理人“`${applyUserId}`”，在代码清单6-1的空启动事件中定义了`activiti:initiator=“applyUserId”`，所以销假节点就可以直接使用变量`applyUserId`的值作为办理人的ID，这样用户任务会自动分配给申请人。

#2处添加了一个字段“销假日期”并指定“`writable`”属性为“`true`”，可以由申请人填写请假的实际结束日期。

在#3处，当申请人销假之后执行表达式“`[execution.setVariable('result', 'ok')]`”，将当前流程实例的变量名“`result`”设置为“`ok`”。

以上情况都是假设部门领导和人事经理都审批通过。根据流程定义可以分析出：如果部门经理和人事经理中的一个审批未通过，则把输出流指定到用户任务“调整申请”，并且该任务的排他分支有两个输出流，一个是“重新申请”，另外一个“结束流程”。代码清单6-5列出了“调整申请”用户任务的XML定义。

代码清单6-5 图6-1请假流程的调整申请用户任务的XML定义

```
<userTask id="modifyApply" name="调整申请" activiti:assignee="${applyUserId}">
  <extensionElements>
    <activiti:formProperty id="startDate" name="请假开始日期" type="date"
      value="${startDate}" datePattern="yyyy-MM-dd" required="true" readable="true" #1
      writable="true"></activiti:formProperty>
    <activiti:formProperty id="endDate" name="请假结束日期" type="date"
      value="${endDate}" datePattern="yyyy-MM-dd" required="true" readable="true" #2
      writable="true"></activiti:formProperty>
    <activiti:formProperty id="reason" name="请假原因" type="string" value="${reason}" #3
      required="true" readable="true" writable="true"></activiti:formProperty>
    <activiti:formProperty id="reApply" name="重新申请" type="enum" required="true" #4
      writable="true">
      <activiti:value id="true" name="重新申请"></activiti:value>
      <activiti:value id="false" name="取消申请"></activiti:value>
    </activiti:formProperty>
  </extensionElements>
</userTask>
<sequenceFlow id="flow10" name="重新申请" sourceRef="exclusivegateway7" #5
  targetRef="deptLeaderAudit">
  <conditionExpression xsi:type="tFormalExpression">
    <![CDATA[${reApply == 'true'}]]></conditionExpression>
  </sequenceFlow>
<sequenceFlow id="flow12" name="结束流程" sourceRef="exclusivegateway7"
  targetRef="endevent1">
  <extensionElements>
    <activiti:executionListener event="take"
      expression="${execution.setVariable('result', 'canceled')}" /> #6
  </extensionElements>
  <conditionExpression xsi:type="tFormalExpression">
    <![CDATA[${reApply == 'false'}]]></conditionExpression>
  </sequenceFlow>
```

第一眼看上去，代码清单6-5和代码清单6-1的XML代码基本一样，但是注意#1、#2、#3处加粗的属性`value`，申请人在启动流程时填写的表单字段可能需要再次修改，所以通过设置`value`属性获取表单字段值。

从#5处开始根据申请人的选择来判断是重新申请还是取消申请（结束流程）。

在#6处，当申请人销假之后执行表达式“`[execution.setVariable('result', canceled)]`”，将当前流程实例的变量名“`result`”设置为“`canceled`”。

6.1.2 单元测试

6.1.1节对整个动态表单的请假流程定义XML做了详细的讲解，本书将对设计的流程定义进行功能性验证，以此判断所设计的流程定义是否能满足实际的需求。

代码清单6-6用单元测试方式验证了在请假流程中部门领导和人事经理审批通过的情况。

代码清单6-6 部门领导和人事经理全部审批通过时的单元测试

```
public class LeaveDynamicFormTest extends AbstractTest {
    @Test
    @Deployment(resources = "chapter6/dynamic-form/leave.bpmn") // 利用注解简化部署
    public void allApproved() throws Exception {
        String currentUserId = "henryyan";
        identityService.setAuthenticatedUserId(currentUserId); // 设置当前用户 #1
        // 启动流程
        ProcessDefinition processDefinition = repositoryService #2-S
            .createProcessDefinitionQuery().processDefinitionKey("leave").singleResult();
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd"); // #2-1
        Map<String, String> variables = new HashMap<String, String>();
        Calendar ca = Calendar.getInstance(); //
        String startDate = sdf.format(ca.getTime()); // 利用日历对象设置
        ca.add(Calendar.DAY_OF_MONTH, 2); // 请假的开始、结束日期
        String endDate = sdf.format(ca.getTime()); //
        variables.put("startDate", startDate);
        variables.put("endDate", endDate);
        variables.put("reason", "公休"); // 从#2-1至此用来设置启动时
        // 需要填写的字段
        // 使用FormService的启动流
        // 程方法
        ProcessInstance processInstance =
            formService.submitStartFormData(processDefinition.getId(), variables); #2-2
        assertNotNull(processInstance); #2-E
        // 部门领导审批通过
        Task deptLeaderTask = taskService.createTaskQuery() #3-S
            .taskCandidateGroup("deptLeader").singleResult();
        variables = new HashMap<String, String>();
        variables.put("deptLeaderApproved", "true"); // 设置审批通过标志 #3-1
        formService.submitTaskFormData(deptLeaderTask.getId(), variables); #3-E
        // 人事审批通过
        Task hrTask = taskService.createTaskQuery().taskCandidateGroup("hr").singleResult(); #4-S
        variables = new HashMap<String, String>();
        variables.put("hrApproved", "true"); // 设置审批通过标志 #4-1
        formService.submitTaskFormData(hrTask.getId(), variables); #4-E
        // 销假（根据申请人的用户ID读取）
        Task reportBackTask = taskService.createTaskQuery() #5-S
            .taskAssignee(currentUserId).singleResult();
        variables = new HashMap<String, String>();
        variables.put("reportBackDate", ca.getTime()); // 设置销假日期
        formService.submitTaskFormData(reportBackTask.getId()); #5-E
        // 验证流程是否已经结束
        HistoricProcessInstance historicProcessInstance = historyService #6-S
            .createHistoricProcessInstanceQuery().finished().singleResult();
        assertNotNull(historicProcessInstance); #6-E
        // 读取历史变量
        Map<String, Object> historyVariables = packageVariables(processInstance); #7-S
        // 验证执行结果
        assertEquals("ok", historyVariables.get("result")); #7-E
    }
    // 读取历史变量
    private Map<String, Object> packageVariables(ProcessInstance processInstance) {
        Map<String, Object> historyVariables = new HashMap<String, Object>();
        List<HistoricDetail> list = historyService.createHistoricDetailQuery()
            .processInstanceId(processInstance.getId()).list();
        for (HistoricDetail historicDetail : list) {
            if (historicDetail instanceof HistoricFormProperty) { // 表单中的字段 #8-S
                HistoricFormProperty field = (HistoricFormProperty) historicDetail;
                historyVariables.put(field.getPropertyId(), field.getPropertyValue());
                System.out.println("form field: taskId=" + field.getTaskId() + ", "
                    + field.getPropertyId() + " = " + field.getPropertyValue());
            } else if (historicDetail instanceof HistoricVariableUpdate) { // 普通变量 #8-E
                HistoricVariableUpdateEntity variable = (HistoricVariableUpdateEntity) historicDetail;
                historyVariables.put(variable.getName(), variable.getValue());
                System.out.println("variable: " + variable.getName() + " = " + variable.getValue());
            }
        }
        return historyVariables;
    }
}
```

对于代码清单6-6，根据6.1.1的分析，整个的测试思路也就清楚了。为了便于逐一分析，后面列出了单元测试的执行结果。

#1处设置的是当前用户的ID，这样当流程在运行过程中需要获取当前操作人时会通过调用identityService.setAuthenticatedUserId()方法来实现，在空启动事件上设置的变量“applyUserId”用来保存启动流程人的ID，从输出结果可以看出applyUserId的变量值为“henryyan”，因此在实际应用中启动流程前调用此方法便显得格外重要。

#2-2处使用了一个新的API，和runtimeService.startProcessInstance()功能类似——启动一个新的流程实例。通过formService.submitStartFormData()方法同样可以启动一个新的流程实例，不过这个方法在执行启动流程实例时会从第二个参数（variables）中读取和表单字段ID相同的项作为表单字段的值，如果某个字段设置了“required”属性并在变量中未找到和表单字段相同的ID或值为空，则抛出异常，并且根据设置的字段类型调用对应的类型转换器来转换字段值。

两者还有一个很重要的区别：通过formService.submitStartFormData()启动流程时字段的值只能为“字符型”（java.lang.String），因为表单的内容都是字符型。

#3-1和#4-1处代码类似，唯一不同的是：设置不同的变量为“true”表示这两个审核节点均通过申请。

#3-E、#4-E、#5-E这三处的功能是提交任务表单并完成任务。功能和taskService.complete()类似，对表单字段的处理和formService.submitStartFormData()方法一样。

#7-S处用来读取历史流程（已经办理过的任务、流程实例）的变量；#7-E处用来验证通过捕获“销假”用户任务的输出流的“take”事件执行的表达式“\${execution.setVariable('result', 'ok')}”的结果是否保存到了数据库，流程结束后从历史变量中查询“result”的值等于“ok”即表示流程正常结束。

#8处根据接口HistoricDetail的实际类型进行了区分，输出结果如下：

```
variable: applyUserId = henryyan
form field: taskId=null, startDate = 2012-11-26
form field: taskId=null, reason = 公休
form field: taskId=null, endDate = 2012-11-28
variable: startDate = Mon Nov 26 00:00:00 CST 2012
variable: endDate = Wed Nov 28 00:00:00 CST 2012
variable: reason = 公休
form field: taskId=19, deptLeaderApproved = true
variable: deptLeaderApproved = true
form field: taskId=26, hrApproved = true
variable: hrApproved = true
form field: taskId=33, reportBackDate = 2012-11-28
variable: reportBackDate = Wed Nov 28 00:00:00 CST 2012
variable: result = ok
```

以“form field:”开头的输出结果是表单字段内容，其他以“variable:”开头的是普通变量。细心的读者可能发现了一个问题，同样的一个变量输出了两次，这是为什么呢？因为在保存表单字段的同时会复制一份经过类型转换的值，并以字段的ID为变量名称保存该变量到数据库，当然前提是要设置引擎的“history”属性为“full”级别。正因如此，在上面的输出结果中，以“form field”开头的变量“startDate”的值为“2012-11-26”（字符型），而以“variable”开头的变量“startDate”的值为一个标准的CST日期。

 **注意** Activiti的history默认级别为audit，要保存流程中产生的变量需要设置引擎的history为full。

在本书配套资源文件中，源码目录的LeaveDynamicFormTest类中还有一个模拟领导审批节点被驳回的例子，申请人在“调整申请”节点选择了“取消申请”，最后变量“result”的值为“canceled”。

6.1.3 在Activiti Explorer中运行流程

通过单元测试模拟请假流程的执行可能比较抽象，在第2章中介绍了Activiti提供的Demo程序——Activiti Explorer，本节将动态表单的请假流程部署到Activiti Explorer中运行，体验一下Web界面的流程驱动。Activiti Explorer的使用教程可以参考2.4.2节。

首先把请假流程的xml和png文件打包为一个zip文件，图6-2展示了打包的请假流程的资源文件。

Name		Date Modified
	leave.bpmn	Today 8:06 PM
	leave.png	Today 8:06 PM
	leave-dynamic-form.zip	Today 9:43 PM

图6-2 请假流程（动态表单）打包结果

在部署“leave-dynamic-form.zip”文件之后，在部署列表可以看到如图6-3所示的列表。

Activiti Explorer

TasksProcesses

DatabaseDeploymentsJobsUsersGroups

Demo processes

leave-dynamic-form.zip



leave-dynamic-form.zip

Deployed moments ago

Process Definitions

[请假流程-动态表单](#)

Resources

[leave.bpmn](#)

[leave.png](#)

图6-3 已部署的请假流程（动态表单）

然后单击“Processes” → “Deployed process definitions” 看到如图6-4所示的请假流程的信息。

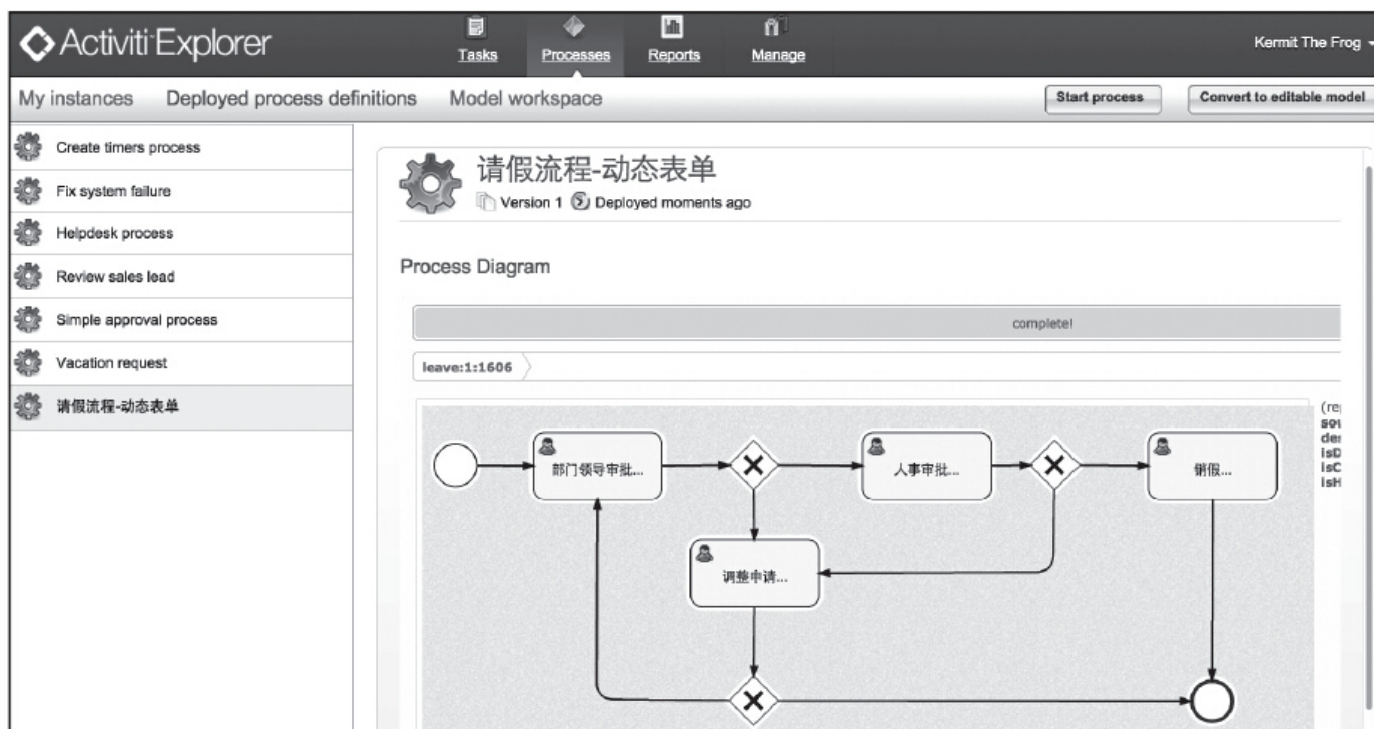
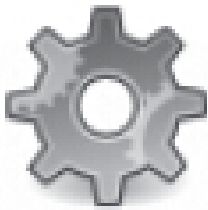


图6-4 已部署的请假流程（动态表单）

要启动请假流程只要单击图6-4右上角的“Start process”按钮即可，显示的界面如图6-5所示的流程启动界面。然后单击在“Processes” → “My instances” 菜单可以看到正在运行的请假流程，如图6-6所示，其中以加粗边框表示当前节点。同时在流程图下方还有任务和变量列表，如图6-7所示，注意变量“applyUserId”的值为“kermit”（使用kermit登录后启动的流程）。



请假流程-动态表单

 Version 1  Deployed 5 minutes ago

请假开始日期 *

2012-11-26



请假结束日期 *

2012-11-28



请假原因 *

公休

Start process

Cancel

图6-5 启动请假流程的界面

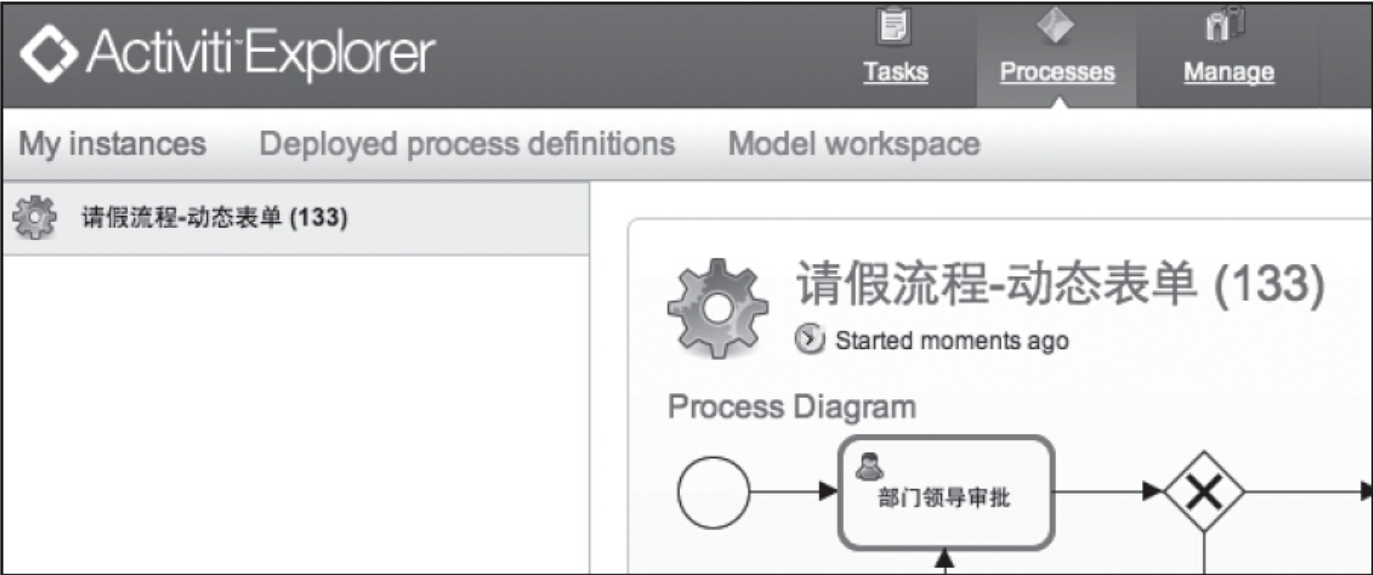


图6-6 运行中的请假流程实例

Tasks

	NAME	PRIORITY	ASSIGNEE	DUE DATE	CREATED	COMPLETED
	部门领导审批	50			moments ago	

Variables

NAME	VALUE
applyUserId	kermit
endDate	Wed Nov 28 00:00:00 CST 2012
reason	公休
startDate	Mon Nov 26 00:00:00 CST 2012

图6-7 流程实例的任务列表和变量列表

接下来就需要处理任务了，但是从图6-8可以看出，所有的统计数字均为0，没有任务需要办理。部门领导的用户任务分配给了角色“deptLeader”，所以要想让kermit可以处理“部门领导审批”任务需要向kermit授予ID为“deptLeader”的角色。

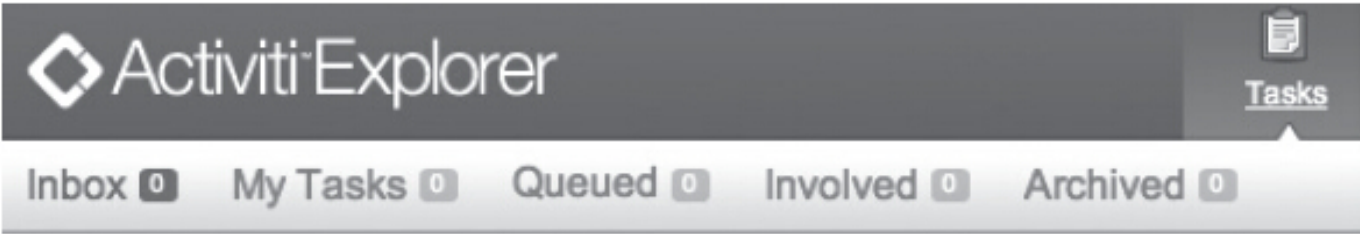


图6-8 kermit在没有被授予部门经理角色时没有任务可以办理

要授予用户kermit角色需要单击菜单“Manage” → “Groups”，然后单击右上角的“Create group”按钮，添加如图6-9和图6-10所示的两个角色。

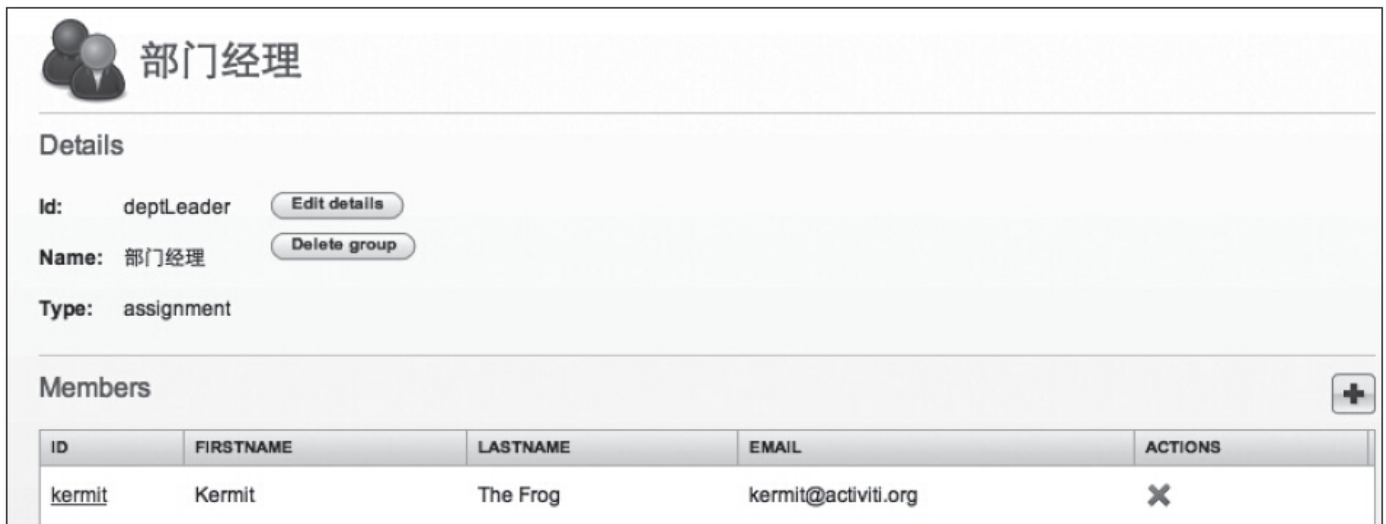


图6-9 添加角色“部门经理”并把用户kermit加入到角色中

把角色“部门经理”和“人事经理”授予kermit之后，单击菜单“Tasks”可以看到“Queued”菜单项显示为“Queued 1”，如图6-11所示。

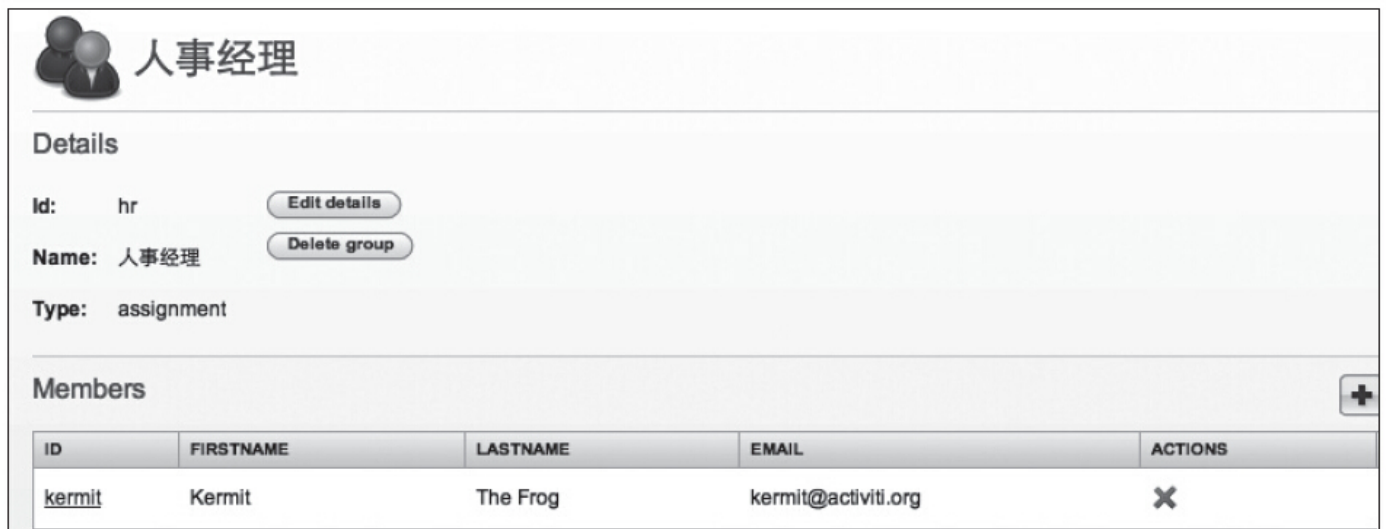


图6-10 添加角色“人事经理”并把用户kermit加入到角色中



图6-11 部门经理有一个待办任务

单击图6-11的“部门经理 (1)”之后可以看到如图6-12所示的任务办理界面，在此界面可以签收 (Claim) 任务、完成任务，当然在完成任务之前需要填写表单信息。

接着单击按钮“Claim”签收任务（此时其他拥有deptLeader角色的人员就没有签收此任务的权限了），执行签收任务之后“审批意见”字段可以编辑了，选择“同意”选项（如图6-12所示），然后单击按钮“Complete task”完成任务办理。

部门领导审批

31

No due date

Medium Priority

Created 16 minutes ago

Claim

This task has no description set.

Part of process: '请假流程-动态表单'

People

No owner

Transfer

No assignee

Reassign

Subtasks

No subtasks defined for this task

Related content

No related content attached for this task

Fill in the form below and complete the task:

请假开始日期

2012-11-26

请假结束日期

2012-11-28

请假原因

公休

审批意见 *

Complete task

Reset form

图6-12 请假流程的“部门领导审批”任务办理界面

此时，因为kermit用户拥有角色“人事经理”，所以刷新页面后可以看到“Queued”中的“人事经理”项显示为1，如图6-14所示。单击“人事经理（1）”后显示如图6-15所示的界面，和图6-13大致类似。

Related content

No related content attached for this task

Fill in the form below and complete the task:

请假开始日期

2012-11-26

请假结束日期

2012-11-28

请假原因

公休

审批意见*

同意

Complete task

Reset form

图6-13 在签收任务之后选择审批意见为“同意”

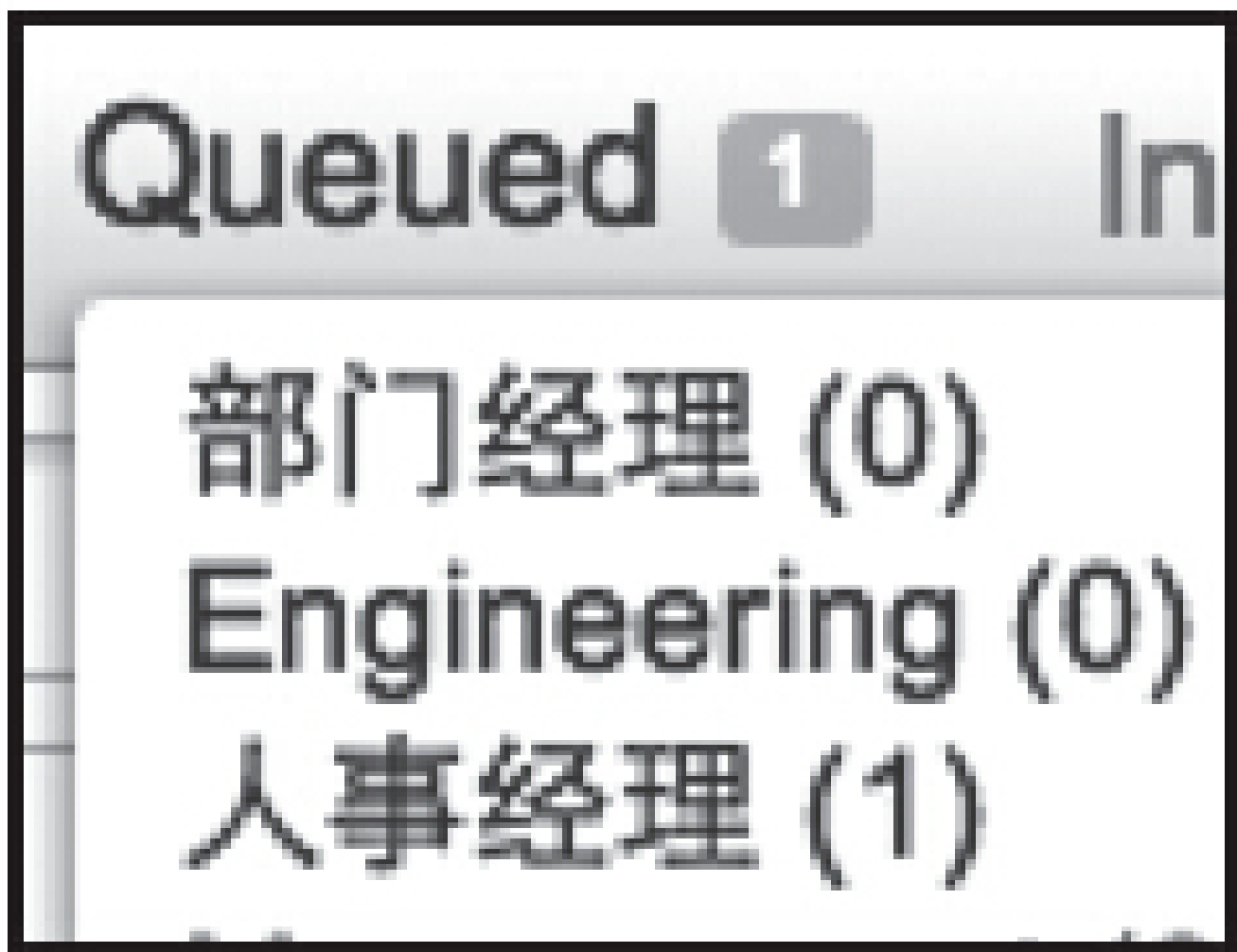


图6-14 人事经理有一个待办任务



人事审批

 No due date

 Medium Priority

 Created 2 minutes ago

Claim

This task has no description set.

Part of process: '请假流程-动态表单'

People

 No owner

Transfer

 No assignee

Reassign

Subtasks

No subtasks defined for this task

Related content

No related content attached for this task

Fill in the form below and complete the task:

请假开始日期



请假结束日期



请假原因

审批意见 *

Complete task

Reset form

图6-15 请假流程的“人事审批”任务办理界面

和处理“部门领导审批”任务一样，选择“审批意见”字段的值为“同意”并点击按钮“Complete task”完成任务办理。

此时再查看图6-7的界面会发现多了两个变量，分别是部门领导和人事经理对申请做出的处理结果，如图6-16所示。

Tasks			
	NAME	PRIORITY	ASSIGNEE
	人事审批	50	 <u>Kermit The Frog</u>
	部门领导审批	50	 <u>Kermit The Frog</u>
	销假	50	 <u>Kermit The Frog</u>
Variables			
NAME		VALUE	
applyUserId		kermit	
deptLeaderApproved		true	
endDate		Wed Nov 28 00:00:00 CST 2012	
hrApproved		true	
reason		公休	
startDate		Mon Nov 26 00:00:00 CST 2012	

图6-16 人事审批任务办理完成后查看运行中流程实例的任务和变量列表

人事经理完成任务之后根据流程图的设计应该流转到“销假”节点，所以此时“Inbox”菜单显示为一个待办任务，如图6-17所示。

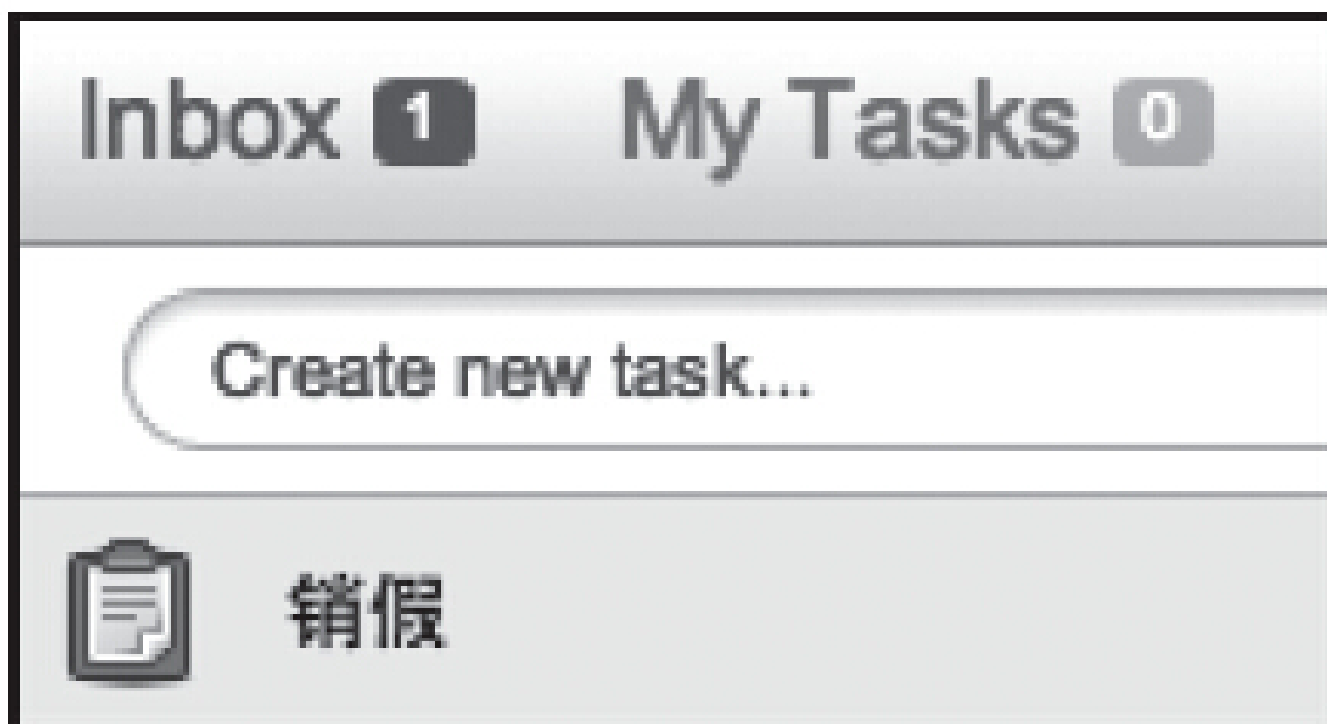


图6-17 人事经理办理完任务后kermit有一个待办任务“销假”

单击图6-17的“销假”任务即可办理任务，如图6-18所示。从图6-18可以看出“销假日期”默认为请假结束日期，因为在流程定义中设置了“销假日期”字段的default属性为“\${endDate}”。单击“Complete task”完成任务办理，至此流程实例结束。



销假

31

No due date

≡

Medium Priority

🕒

Created moments ago

This task has no description set.

Part of process: '请假流程-动态表单'

People



No owner

Transfer

Subtasks

No subtasks defined for this task

Related content

No related content attached for this task

Fill in the form below and complete the task:

请假开始日期

2012-11-26

📅

请假结束日期

2012-11-28

📅

请假原因

公休

销假日期*

2012-11-28

📅

Complete task

Reset form

图6-18 请假流程的“销假”用户任务

 **提示** 读者在执行过程中可以随时查看“Manage”→“Database”中的Activiti数据库表，这样能够方便地了解在执行某个操作后数据库会有哪些变化，以便更好地理解执行过程。

流程的“调整申请”用户任务在“部门领导审批”或者“人事审批”选择“拒绝”之后会被创建，和销假一样会把用户任务自动分配给申请人，可以自行模拟一次被拒绝的情况，然后在“调整申请”节点调整后向部门经理提交申请。

6.2 实现自己的Activiti Explorer

Activiti Explorer是Activiti提供的一个简易的Demo程序，在6.1.3节使用Activiti Explorer部署了动态表单的请假流程定义并顺利完成了流程实例的执行。本章主要介绍如何运用动态表单和外置表单，而6.1.3节仅仅告诉读者如何使用Activiti Explorer对动态表单的支持，那么Activiti Explorer具体是如何实现的呢？本节将一探究竟。

这里明确一下实际需求，重新设计的Activiti Explorer的目的有以下几点：

- 学习如何根据用户任务的表单字段动态生成表单。
- 弥补官方的Activiti Explorer不支持外置（FormKey）表单的缺陷。
- 作为后续章节学习的一个基础组件。

6.2.1 完善身份验证功能

为了模拟一个真实的系统功能，在第5章的基础上添加了登录和退出功能，如图6-19和6-20所示，在图6-19中输入用户名和密码之后单击“登录系统”按钮后打开如图6-20所示的页面，可以单击右上角的下拉框，然后单击“安全退出”切换用户。



图6-19 登录页面



图6-20 系统首页

本章内容的源码位于本书配套资源文件的“chapter6-oa-forms”文件夹中，可以导入到Eclipse或其他IDE中运行、查看。

请假流程涉及多个角色（Activiti中称之为“组”，即Group），这里假定有如下几个：

- 总经理
- 部门领导
- 人事经理

同时涉及技术部、人事部、业务部、总经理室三个部门。表6-2列出了每个用户的所属部门及拥有的角色。

表6-2 每个用户所属部门及拥有的角色

登 录 名	姓 名	所 属 部 门	拥 有 角 色
eric	Eric Li	市场部	
tom	Tom Wang	市场部	
kermit	Kermit Miao	市场部	部门经理
amy	Amy Zhang	业务部	
andy	Andy Zhao	业务部	部门经理
bill	Bill Zheng	总经理室	总经理
jenny	Jenny Luo	人事部	人事经理
henry	Henry Yan	IT 部	系统管理员

在启动应用（在终端中输入mvn jetty:run命令回车）之前需要先初始化数据库，把表6-2中的用户及角色数据插入到Activiti数据库中，此操作通过Maven的antrun^[1]插件执行：

```
mvn antrun:run -Pinit-db
```

其中的init-db是在pom.xml中定义的一个Profile^[2]，在此Profile中定义了初始化的操作，使用Dbunit^[3]工具把“chapter6-oa-forms/src/main/resources/data/identity-data.xml”中的数据通过JDBC方式插入到数据库中。

为了让我们的Activiti Explorer看起来接近实际的企业应用，利用Bootstrap框架设计了一个比较简洁的页面框架，并借用第5章实现的流程定义列表作为本章内容的实现基础，图6-21为整合后的界面。

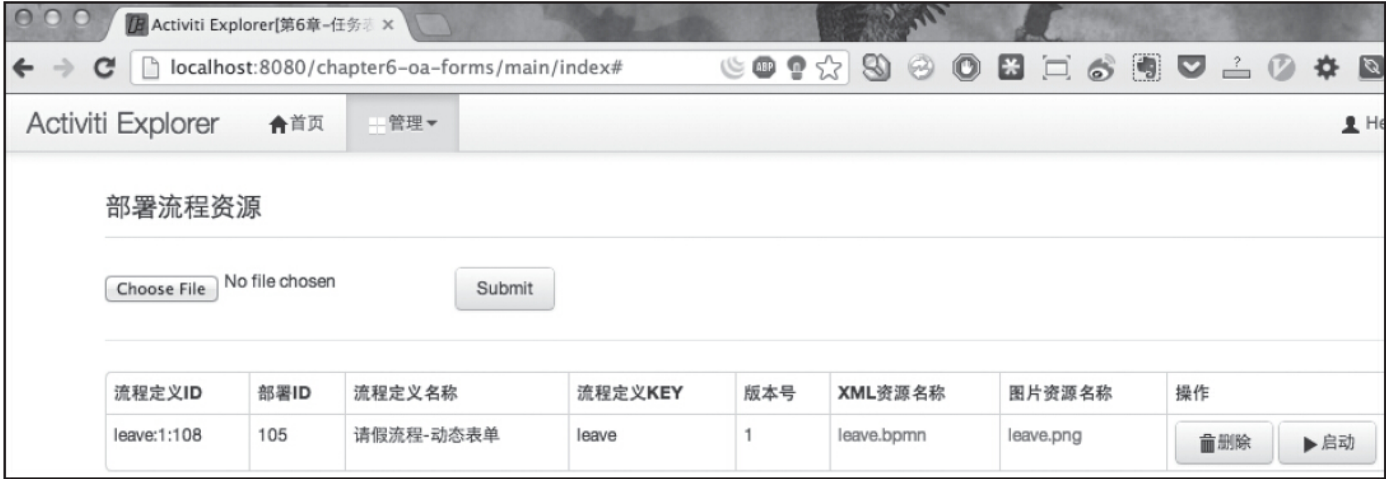


图6-21 添加了下拉菜单的Activiti Explorer

6.2.2 流程启动表单

图6-21的列表中添加了一个“启动”按钮用来启动一个流程实例。在官方的Activiti Explorer中启动流程实例是从已部署的流程定义列表中选择一个流程定义，然后单击“Start process”按钮，页面刷新后会显示流程的空启动事件定义的表单字段，所以我们第一步要做的就是读取流程启动事件的表单。

1.读取流程启动表单

单击图6-21中列表的“启动”按钮即可读取空启动事件的表单字段，如图6-22所示。然后单击“启动流程”按钮。

Activiti Explorer

首页

管理

启动流程—请假流程-动态表单

请假开始日期:

2012-11-08

请假结束日期:

2012-11-15

请假原因:

公休

▶ 启动流程

图6-22 启动流程填写表单

看似简单的两步操作，代码是如何实现的呢？为了实现这两步，我们定义了一个类“Process-DefinitionController”负责接收并处理请求，其中有两个方法分别负责读取启动流程的表单和启动流程。

图6-21中的“启动”按钮的功能实际是跳转到一个链接，代码如下：

```
<a href='${ctx }/chapter6/getform/start/${pd.id }'>启动</a>
```

其中的“\${pd.id}”即第5章的流程定义列表中流程定义对象的ID，本章的例子只是基于第5章的列表进行了扩展而已。

代码清单6-7展示了“启动”按钮对应的后台处理代码。

代码清单6-7 读取启动流程表单

```
@Controller
@RequestMapping(value = "/chapter6") #1
public class ProcessDefinitionController extends AbstractController {
    // 读取启动流程的表单字段
    @RequestMapping(value = "getform/start/{processDefinitionId}") #2
    // processDefinitionId参数对应链接的最后一个参数
    public ModelAndView readStartForm(
        @PathVariable("processDefinitionId") String processDefinitionId) throws Exception {
        String viewName = "chapter6/start-process-form"; // 位于WEB-INF/views目录
        ModelAndView mav = new ModelAndView(viewName);
        StartFormData startFormData = formService.getStartFormData(processDefinitionId); #3
        mav.addObject("startFormData", startFormData); #4-S
        mav.addObject("processDefinitionId", processDefinitionId); #4-E
        return mav;
    }
}
```

首先说明一下本章所有新增的类访问路径均以“/chapter6”开头，例如代码清单6-7的#1处。

#2处声明readStartForm()方法的访问路径为“getform/start/{processDefinitionId}”。

#3处是代码清单6-7的重点，通过getStartFormData()方法即可读取启动流程时需要填写的表单数据（设计流程定义时定义的字段集合），得到一个StartFormData对象，还可以获取表单字段对象集合，并且可以读取表6-1中的所有属性值。

#4处把需要的对象设置到ModelAndView对象中，以便在展示层可以利用。

有了读取表单内容的功能支持就可以编写对应的展示层了。依然使用JSP作为展示层的实现，功能就是解析表单数据把不同的字段类型转换为HTML代码（图6-22中的三个输入框；笔者使用了Twitter的Bootstrap^[4]作为CSS基础框架来渲染页面元素）。

代码清单6-8列出了图6-22的JSP实现代码，读者可以在本书配套资源文件的chapter6-oa-forms/src/main/webapp/WEB-INF/views/chapter6目录中找到JSP文件“start-process-form.jsp”。

代码清单6-8 根据StartFormData对象转换为HTML元素

```
<form method="post"
action='${ctx }/chapter6/process-instance/start/${processDefinitionId}'> #1
<c:forEach items='${startFormData.formProperties}' var="fp">
```

```
<c:set var="fpo" value="${fp}" /> // 把${fp}存放到pageContext对象的属性中
<% // 把需要获取的属性读取并设置到pageContext域
// 要获取日期字段的格式需要调用org.activiti.engine.impl.form.AbstractFormType类的
// getInformation()方法，但是EL不支持直接调用方法，所以不得不转化
FormType type = ((FormProperty)pageContext.getAttribute("fpo")).getType();
String[] keys = {"datePattern"}; // 定义需要读取的其他扩展属性的名称
for (String key: keys) {
    pageContext.setAttribute(key, ObjectUtils.toString(type.getInformation
        (key))); // #2
} %>
<%-- 文本或者数字类型 --%>
<c:if test="${fp.type.name == 'string' || fp.type.name == 'long'}"> // #3-S
    <label for="${fp.id}">${fp.name}</label>
    <input type="text" id="${fp.id}" name="${fp.id}" />
</c:if> // #3-E
<c:if test="${fp.type.name == 'date'}"> <%-- 日期 --%>
    <label for="${fp.id}">${fp.name}</label>
    <input type="text" id="${fp.id}" name="${fp.id}" class="datepicker"
        data-date-format="${fn:toLowerCase(datePattern)}" /> // #4
</c:if>
</c:forEach>
<button type="submit" class="btn"><i class="icon-play"></i>启动流程</button> // #5
</form>
```

代码清单6-8用JSP和HTML代码联合起来完成解析表单字段的功能，#1处指定了表单的action属性，即单击了#5处的“启动流程”按钮后请求的URL。

在#2处从类FormType的对象type中读取字段的扩展属性（只有所有字段共有的字段才能通过getXxx()方法获取，所以id、name属性可以直接通过EL表达式获取）并设置到pageContext的属性中，这样在#4处就可以通过EL表达式的方式获取日期类型字段的日期格式。

#3处用来处理type为string、long的字段，这里仅输出了一个普通的文本框组件的HTML代码。在#4处的日期组件也与此类似，只不过针对日期类型添加了一个属性“data-date-format”用来保存日期的格式（一个基于Bootstrap的日期插件规范要求）。

因此在单击图6-21的“启动”按钮后即可显示图6-22所示的表单内容。

在6.1.2节的单元测试中用runtimeService.startProcessInstanceByKey()的方式启动了流程，将变量通过一个Map对象封装传递给方法。本节的内容是基于Web页面的方式操作，但是万变不离其宗，本例同样也要把表单的值封装成一个Map对象传递给启动流程的方法。

图6-22的表单提交后访问一个后台的服务，在ProcessDefinitionController.java类中定义了一个方法startProcessInstance(String processDefinitionId)，见代码清单6-9。

代码清单6-9 读取表单数据并启动流程实例

```
@RequestMapping(value = "process-instance/start/{processDefinitionId}")
public String startProcessInstance(@PathVariable("processDefinitionId")
    String processDefinitionId, HttpServletRequest request) {
    // 先读取表单字段，再根据表单字段的ID读取请求参数值
    StartFormData formData = formService.getStartFormData(processDefinitionId); // #1-S
    // 从请求中获取表单字段的值
    List<FormProperty> formProperties = formData.getFormProperties(); // #1-L
    Map<String, String> formValues = new HashMap<String, String>();
    for (FormProperty formProperty : formProperties) {
        String value = request.getParameter(formProperty.getId()); // 从request读取参数值
        formValues.put(formProperty.getId(), value); // 把表单字段及对应值设置到Map对象中
    }
    // 获取当前登录的用户
    User user = UserUtil.getUserFromSession(request.getSession()); // #2-S
    identityService.setAuthenticatedUserId(user.getId());
    // 提交表单字段并启动一个新的流程实例
    ProcessInstance processInstance =
        formService.submitStartFormData(processDefinitionId, formValues); // #2-E
    return "redirect:/chapter5/process-list";
}
```

代码清单6-9的功能是读取表单参数并启动一个新的流程实例，分为两个步骤：

- #1处为第一个步骤，读取启动表单的字段（#1-S处），在#1-L处读取表单字段集合，最后循环表单字段并把请求的表单字段值用Map方式保存。
- #2处为第二个步骤，启动流程实例后返回流程定义列表。

6.2.3 任务签收与办理

如何读取任务在前面的章节中已经多次涉及，现在需要使用Web方式读取待办任务。待办任务的数据来源有以下4类。

- 第一种是直接分配：直接通过humanPerformer元素或activiti:assignee属性指定某个人的ID作为任务办理人。
- 第二种是在候选人范围之内。
- 第三种是属于某个候选组。
- 第四种是代办的任务：这类任务的来源相对来说比较特殊。我们用模拟场景的方式说明这一概念：有一个属于用户A的任务，A把任务交给用户B办理（此时B是办理人，A是任务所有人（Owner）），B办理完之后又将任务回归到A，这个过程就称为代办（在第11章中将详细讲解）。这种功能常用于上级的任务交给下级代办，或者当某个用户因为其他原因没时间处理任务时将其交由其他人代办。

和任务相关的另外一个概念“转办”，读者也需要了解：把一个任务的办理人重新分配给其他的用户的过程称为转办。

一个简单的任务列表包含任务对象的几个属性，如图6-23所示。

任务ID	任务名称	流程实例ID	任务创建时间	办理人	操作
215	部门领导审批	205	Tue Nov 27 22:41:46 CST 2012		

图6-23 一个简单的任务列表

在启动了流程之后如果在任务列表没有任务要退出，则使用 “kermit” 用户登录，因为 “部门领导审批” 任务需要具有部门领导角色的用户才有权限签收与办理。

负责任务相关功能的控制器类名为TaskController，其中的todoTasks()方法负责读取任务列表。代码清单6-10列出了实现读取待办任务列表的todoTasks()的代码片段。

代码清单6-10 读取待办任务列表的代码片段

```
ModelAndView mav = new ModelAndView(viewName);
User user = UserUtil.getUserFromSession(session);
// 读取直接分配给当前人或者已经签收的任务
List<Task> doingTasks = taskService.createTaskQuery().taskAssignee(user.getId()).list(); #1
// 等待签收的任务
List<Task> waitingClaimTasks = taskService.createTaskQuery(). // TaskQuery对象
    .taskCandidateUser(user.getId()).list(); #2
// 合并两种任务
List<Task> allTasks = new ArrayList<Task>();
allTasks.addAll(doenTasks);
allTasks.addAll(waitingClaimTasks);
mav.addObject("tasks", allTasks);
```

代码清单6-10的#2处需要特别说明一下：taskCandidateUser(user.getId())方法读取的任务列表是待办任务数据来源的前两种的并集。

在调用TaskQuery对象的taskCandidateUser()方法后查询结果包含任务数据来源的第二种和第三种情况，首先查询直接用activiti:cadidateUser指定的用户集合，其次查询activiti:cadidateGroups属性指定组，在查询组时会根据用户与组的关系联合查询。

在Activiti 5.16版本中添加了一个专门用来查询这两种任务集合的过滤器，名称为taskCandidateOrAssigned（userId），在Activiti 5.16之后的版本中使用该过滤器可以简化待办任务的查询方式。代码清单6-10中查询待办任务的代码块可以用下面的一行代码取代：

```
List<Task> tasks = taskService.createTaskQuery()
    .taskCandidateOrAssigned(user.getId()).list();
```

图6-23的列表是根据代码清单6-10的任务集合循环输出的。代码清单6-11列出了输出列表的部分EL表达式代码。

代码清单6-11 图6-23中列表的JSP代码

```
<td>${task.id}</td> // 可以查看API获取Task的更多属性
<td>${task.name}</td>
<td>${task.processInstanceId}</td>
<td>${task.createTime}</td>
<td>${task.assignee}</td>
<td> // 当办理人属性为空时任务需要签收
<c:if test='${empty task.assignee}'><a class="btn" href="claim/${task.id}">签收</a></c:if>
<c:if test='${not empty task.assignee}'> // 当办理人属性不为空时可以办理任务
    <a class="btn" href="getform/task/${task.id}"><i class="icon-user"></i>办理</a>
</c:if></td>
```

和代码清单6-6中处理任务的方式一样，首先需要部门领导签收此任务，也就是单击图6-23中的 “签收” 按钮，然后刷新页面显示如图6-24所示的列表，原来的 “签收” 按钮改成了 “办理” 按钮。

任务ID	任务名称	流程实例ID	任务创建时间	办理人	操作
215	部门领导审批	205	Tue Nov 27 22:41:46 CST 2012	kermit	

图6-24 单击 “签收” 按钮之后显示 “办理” 按钮

用户的任务是和人交互，所以当单击 “办理” 按钮时需要读取任务的表单。读取任务表单的方式和读取启动流程表单类似，单击 “办理” 按钮后读取的任务表单并自动填充了申请人填写的信息，如图6-25所示。

任务办理—部门领导审批

请假开始日期:2012-11-28

请假结束日期:2012-11-30

请假原因:公休

审批意见:拒绝

▶ 完成任务

图6-25 单击图6-24中的“办理”按钮后显示的任务办理页面

代码清单6-12列出了读取任务表单的Java代码，和读取启动流程的表单类似。

代码清单6-12 读取用户任务表单

```
@RequestMapping(value = "task/getform/{taskId}")
public ModelAndView readTaskForm(@PathVariable("taskId") String taskId) throws Exception {
    ModelAndView mav = new ModelAndView("chapter6/task-form");
    TaskFormData taskFormData = formService.getTaskFormData(taskId);    #1
    mav.addObject("taskFormData", taskFormData);
    return mav;
}
```

代码清单6-12通过formService.getTaskFormData()方法读取一个任务的表单对象，然后把表单数据交给视图“chapter6/task-form”处理，即得到了如图6-25所示的页面。其中“WEB-INF/views/chapter6/task-form.jsp”和代码清单6-8类似，故不再列出，读者可以打开源码查看如何根据表单字段的属性设置HTML元素的属性。

图6-25方框中的三个字段是灰色的，表示只读，只有“审批意见”字段可以编辑，单击按钮“完成任务”提交完成任务请求。查看图6-25的页面源码可以看到表单的实际路径如下：

```
<form action="/chapter6-oa-forms/chapter6/task/complete/451" method="post">
```

其中form的action属性中的“451”是用户任务“部门领导审批”的ID，对应的后台处理代码见代码清单6-13。

代码清单6-13 读取完成任务的表单值并完成任务

```
@RequestMapping(value = "task/complete/{taskId}")
public String completeTask(@PathVariable("taskId") String taskId, HttpServletRequest request) throws Exception {
    TaskFormData taskFormData = formService.getTaskFormData(taskId);    #1
    // 从请求中获取表单字段的值
    List<FormProperty> formProperties = taskFormData.getFormProperties();
    Map<String, String> formValues = new HashMap<String, String>();
    for (FormProperty formProperty : formProperties) {
        if (formProperty.isWritable()) { // 排除当前任务表单中不可写的字段    #2
            String value = request.getParameter(formProperty.getId());
            formValues.put(formProperty.getId(), value);
        }
    }
    // 提交用户任务表单并完成任务
    formService.submitTaskFormData(taskId, formValues);    #3
    return TASK_LIST;
}
```

代码清单6-13中的实现方式和启动流程类似，在#1处首先读取任务表单，然后从request对象中获取表单字段的值。

#2处是新添加的代码，用来排除当前任务表单中不可写的字段。如果不排除不可写的字段而在formValue变量中添加了属性，那么Activiti会抛出异常提示字段不可写。

单击了图6-25的“完成任务”按钮后切换用户“jenny”可以看到如图6-26所示的任务列表，有一条“人事审批”的任务等待签收。

流程中剩余的其他任务不再一一列举，读者可以选择不同的审批意见来模拟不同的处理情况。

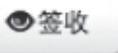
任务ID	任务名称	流程实例ID	任务创建时间	办理人	操作
470	人事审批	441	Wed Nov 28 01:21:06 CST 2012		

图6-26 任务“人事审批”等待办理

6.2.4 自定义表单的字段类型

需要为动态表单的每个字段指定一个类型，目前Activiti默认支持的类型有string、long、enum、date、boolean、collection。在实际的应用中，如果需要其他类型的字段该如何处理呢？很幸运，Activiti允许自定义表单字段类型，比如执行一段Javascript脚本。

在启动流程的表单中添加一段Javascript代码，用于提示表单加载完毕。为了演示如何自定义表单字段，在原来的请假流程基础上进行了更改，又重新部署了一个版本号为2（如果流程ID相同，引擎会自动累加版本号）的请假流程，如图6-27所示。

流程定义ID	部署ID	流程定义名称	流程定义KEY	版本号
leave:1:204	201	请假流程-动态表单	leave	1
leave:2:604	601	请假流程-动态表单	leave	2

图6-27 部署版本号为2的请假流程

首先需要在空启动事件的表单最后（只有在其他的HTML元素都渲染完之后执行脚本才符合逻辑）添加自定义的表单字段。代码清单6-14列出了自定义的Javascript表单字段。

代码清单6-14 自定义的Javascript表单字段

```
<startEvent id="startevent1" name="Start" activiti:initiator="applyUserId">
  <extensionElements>
    <activiti:formProperty id="validScript" type="javascript" #1
      default="alert('表单已经加载完毕');"> #2
    </activiti:formProperty>
  </extensionElements>
</startEvent>
```

代码清单6-14的#1处设置了属性“type”为“javascript”，#2处的“default”属性设置了一段Javascript脚本。读者可能会问：为什么使用default属性而不是value？当读取启动表单字段时，还未启动流程引擎会使用default属性作为value属性的值，所以在代码清单6-15中使用“\${fp.value}”读取Javascript表单字段的脚本内容。

代码清单6-15 读取自定义的Javascript表单字段

```
<c:if test="${fp.type.name == 'javascript'}">
  <script type="text/javascript">${fp.value}</script>
</c:if>
```

在流程定义文件中定义了自定义的表单字段后，还需要让引擎加载自定义表单解析类，用来转换表单值，所有自定义的表单字段需要继承一个表达类型抽象类“org.activiti.engine.impl.form.AbstractFormType”。Javascript表单字段的类JavascriptFormType如代码清单6-16所示。

代码清单6-16 用于转换Javascript表单字段的类

```
public class JavascriptFormType extends AbstractFormType {
  @Override
  public String getName() {
    return "javascript"; // 和流程定义文件的type字段匹配 #1
  }
  @Override
  public Object convertFormValueToModelValue(String propertyValue) { #2
    return propertyValue; // 把表单填写的内容转换为Java的对象类型
  }
  @Override
  public String convertModelValueToFormValue(Object modelValue) { #3
    return (String) modelValue; // 把Java对象类型转换为字符串
  }
}
```

代码清单6-16展示了一个标准的表单字段类型的转换类，首先在#1处的getName()方法中设置唯一的表单字段类型标示符，在#2和#3处分别处理字符串与Java对象类型的互相转换。本例的Javascript是纯文本，所以不需要做其他特殊处理，但是对于日期类型的字段就需要使用日期格式化类进行互相转换，读者可以参考引擎源码的日期表单字段类型的类“org.activiti.engine.impl.form.DateFormType”。

自定义的表单字段类型需要在流程引擎中进行注册，否则引擎会提示未找到对应的类型转换类。注册表单字段类型的配置如代码清单6-17所示。

代码清单6-17 注册自定义表单字段类型

```
<bean id="processEngineConfiguration"
  class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">
  <property name="customFormTypes">
    <list>
      <bean class="me.kafeitu.activiti.chapter6.form.JavascriptFormType" />
    </list>
  </property>
</bean>
```

```
</list>
</property>
</bean>
```

在代码清单6-17中通过引擎的属性“customFormTypes”设置了List集合并添加了一个Bean对象，这样在引擎运行过程中遇到“javascript”类型的字段时就会调用自定义表单类型进行类型转换。

现在再次部署流程定义“请假流程-动态表单”的Javascript版本启动后就会弹出对话框，如图6-28所示。



图6-28 表单渲染完成后执行Javascript脚本的效果

至此，关于应用动态表单的内容画上了句号，从6.1和6.2节中我们学会了如何设计动态表单的定义，如何读取流程启动表单并启动流程实例，如何读取任务表单并完成任务。

[1] maven-antrun-plugin允许通过Maven执行ant任务（target），插件主页：<http://maven.apache.org/plugins/maven-antrun-plugin/>

[2] Profile是Maven的一个独特功能。对于一个相同的项目来说，使用不同的Profile可以产生不同的结果。一般用来区分开发环境、生产环境，因为这两个环境的配置文件不同（例如数据库、日志等），在一个Profile中可以定义需要额外执行的插件、命令等。有兴趣的读者可以参考许晓斌著《Maven实战》的Profile章节。

[3] Dbunit常用于测试阶段，也可以作为数据库的转换工具使用，可以把xml格式的数据转换为sql，然后通过jdbc持久化到数据库。官网地址：<http://www.dbunit.org/>。

[4] Bootstrap,由Twitter开发的开源CSS框架，官网地址：<http://twitter.github.com/bootstrap>。

6.3 外置表单

动态表单在页面上的显示方式是逐行输出，这样在字段较多的情况下会导致表单非常长，用户体验非常差，而且会浪费很多空间（右侧是空白的），使页面的显示非常死板单调。此外，如果需要除引擎默认支持的字段类型之外的字段，还需要在抽象类的基础上进行自行扩展。

动态表单的字段以键值对形式存储在引擎的变量表（ACT_HI_DETAIL）中，这对于数据结构比较简单的业务来说还能满足，但是对于业务关联关系比较复杂且日后需要数据分析的情况就显得力不从心了。

外置表单可以很好地解决动态表单的不足，因为在流程运行时表单内容（从部署的表单文件中读取）会原样显示在页面，当然同样支持动态表单的字段值的自动填充功能。

因此，根据外置表单的特点，在实际应用中大多数应用利用都是外置表单，流程中的一个或多个用户任务对应一个表单文件（扩展名为.form）。还有一部分应用使用的是普通表单，即把表单内容直接写在展示层页面（例如Jsp、Jsf、html等文件）中，将在7.5节详细讲解。

6.3.1 流程定义

本节还是以请假流程为例，在保留所有业务条件的情况下把表单字段元素统统删除，取而代之的是在启动事件和每个用户任务上设置表单属性“activiti:formKey”（参考表4-1）。外置表单版本的请假流程定义见代码清单6-18，其中省略了一些输出流的xml，读者可以参考动态表单版本的请假流程定义。

代码清单6-18 外置表版本的请假流程定义

```
<process id="leave-formkey" name="请假流程--外置表单">
  <startEvent id="startevent1" name="Start" activiti:initiator="applyUserId"
    activiti:formKey="chapter6/leave-formkey/leave-start.form"></startEvent>
  <userTask id="deptLeaderVerify" name="部门经理审批"
    activiti:candidateGroups="deptLeader"
    activiti:formKey="chapter6/leave-formkey/ approve.form"/>
  <userTask id="hrVerify" name="人事经理审批" activiti:candidateGroups="hr"
```

```

    activiti:formKey=" chapter6/leave-formkey/approve.form"></userTask>
<userTask id="reportBack" name="销假" activiti:assignee="${applyUserId}"
    activiti:formKey=" chapter6/leave-formkey/report-back.form"></userTask>
<userTask id="modifyApply" name="调整申请内容" activiti:assignee="${applyUserId}"
    activiti:formKey=" chapter6/leave-formkey/modify-apply.form"></userTask>
<endEvent id="endEvent1" name="End"></endEvent>
</process>

```

代码清单6-18中的空启动事件与4个用户任务均使用属性`activiti:formKey`设置表单名称。值得一提的是`activiti:formKey`属性支持动态设置（即使用变量方式设置），例如`activiti:formKey="${fooFormName}.form"`，在任务启动或完成时通过变量设置`fooFormName`即可动态设置表单内容，这对根据业务状态读取不同表单非常有用。

注意，每个`activiti:formKey`属性都以“chapter6/leave-formkey/”开头，表示包的路径。图6-29展示了外置表单资源内容的目录结构。

其中流程定义文件和表单内容存放在`src/main/resources/chapter6/leave-formkey`包中，对应的单元测试是`src/test/java/me/kafeitu/activiti/LeaveFormKeyTest.java`。

在部署外置表单流程时需要把流程定义文件和表单文件打包部署，因为当读取表单（启动事件和用户任务）时会以当前流程的部署ID和`activiti:formKey`属性为过滤条件，所以当部署了同名的form文件时多个流程定义或相同流程不同版本间也不会冲突。

在流程定义文件中指定的form文件的名称需要在部署包（压缩文件）中保持相同的层级。和Java的jar包类似，源码中的路径和jar文件中class所在的路径相同。图6-30展示了压缩文件`leave-formkey.zip`的目录结构，和图6-29方框内的资源路径匹配。

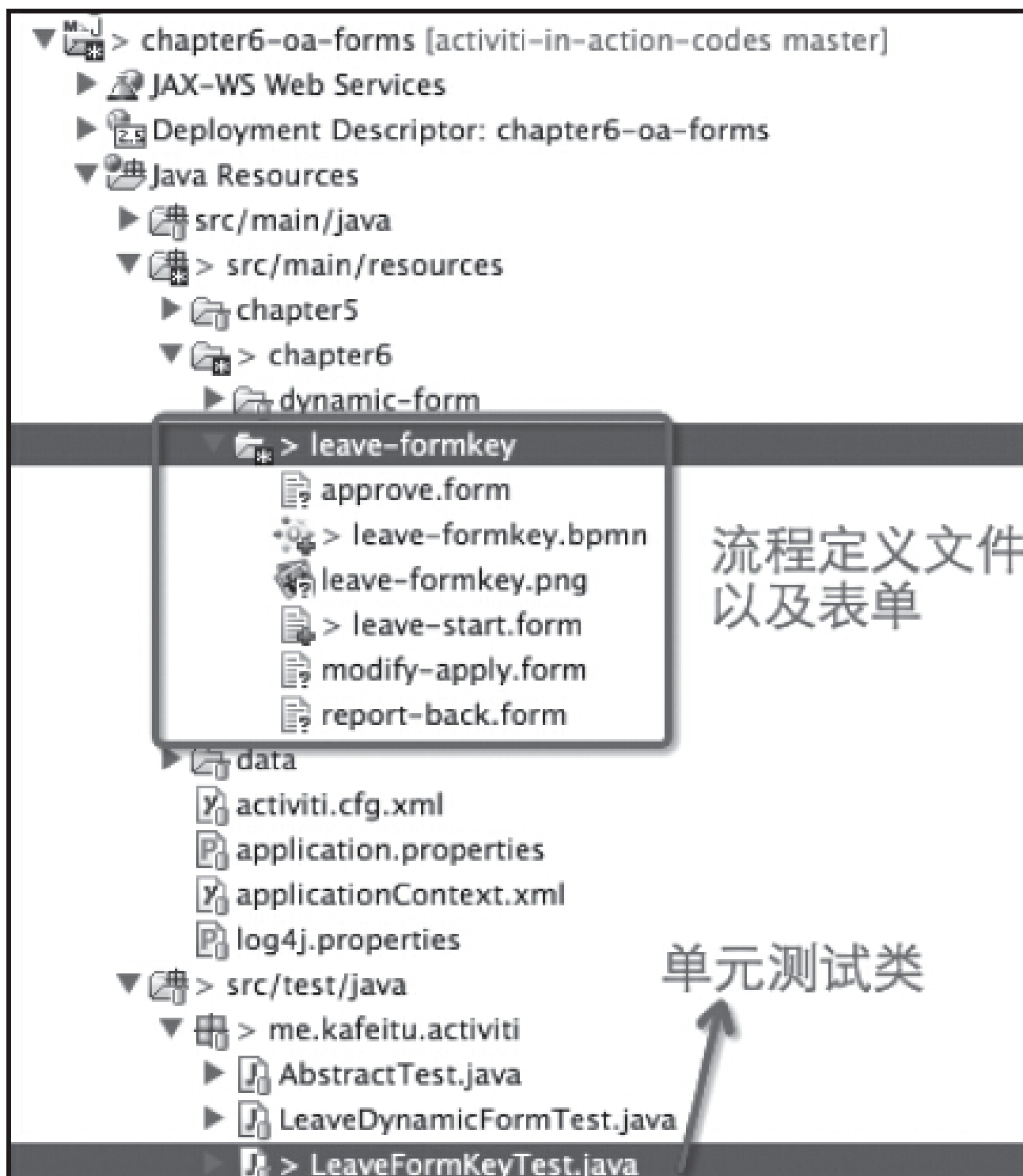


图6-29 外置表单资源的目录结构

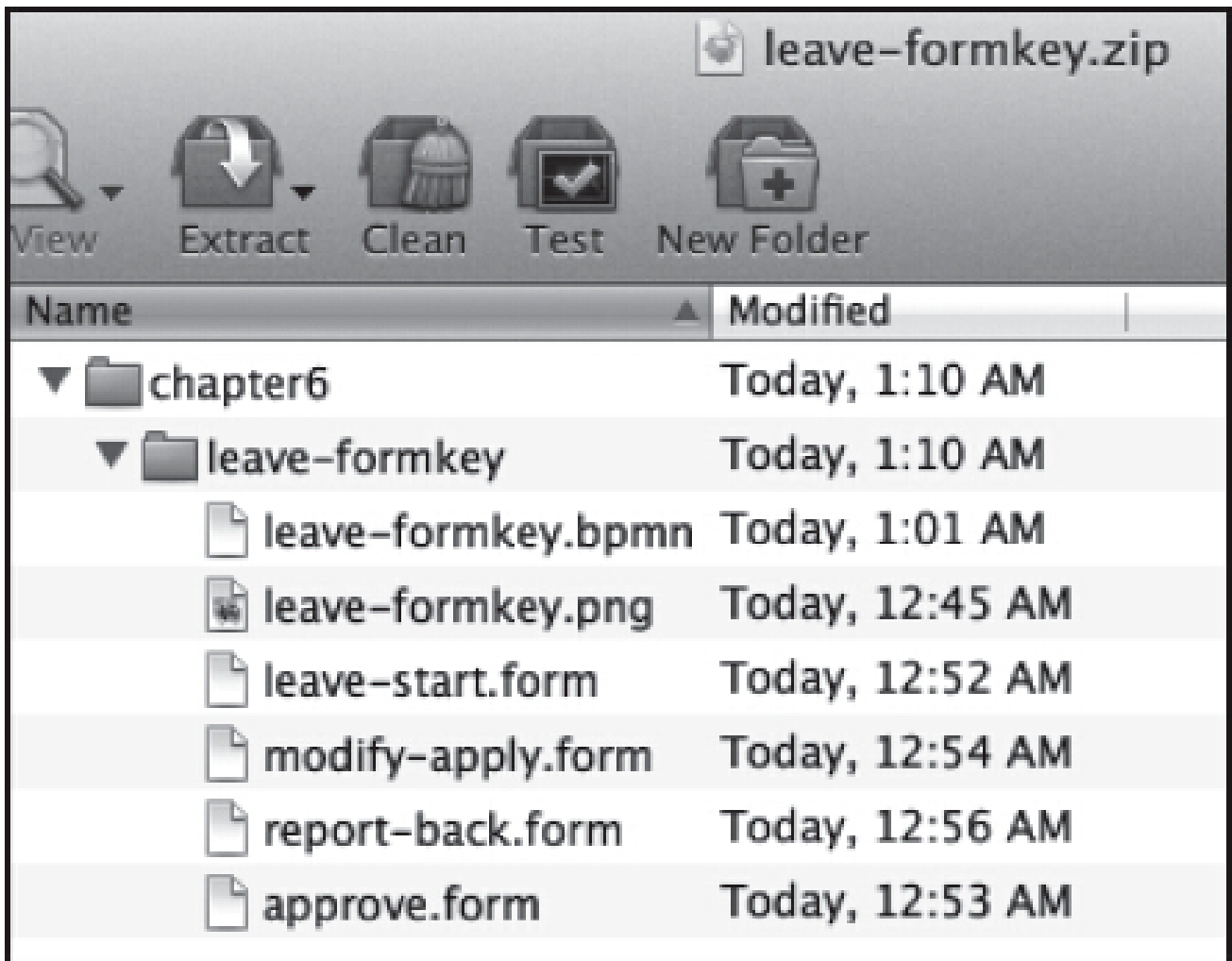


图6-30 压缩文件leave-formkey.zip文件的目录结构

6.3.2 单元测试

外置表单和动态表单的API仅在读取表单内容上有区别，提交表单内容的API和动态表单相同，都是以键值对方式保存表单字段值，如代码清单6-19所示。

代码清单6-19 外置表单的单元测试

```
public class LeaveFormKeyTest extends AbstractTest {
    @Test
    @Deployment(resources = { "chapter6/leave-formkey/leave-formkey.bpmn",
        "chapter6/leave-formkey/leave-start.form", // 流程表单和流程定义文件
        "chapter6/leave-formkey/approve.form", // 一同部署，否则引擎会提示
        "chapter6/leave-formkey/report-back.form", // 找不到表单文件
        "chapter6/leave-formkey/modify-apply.form" })
    public void allPass() throws Exception {
        http://www.hzcourse.com/resource/readBook?path=/openresources/teach_ebook/uncompressed/15030/OEBPS/Text/... // 省略了部分代码
        // 读取启动表单
        Object renderedStartForm =
            formService.getRenderedStartForm(processDefinition.getId()); // #2
        ... // 省略了部分代码
        // 部门领导审批通过
        Task deptLeaderTask = taskService.createTaskQuery()
            .taskCandidateGroup("deptLeader").singleResult();
        assertNotNull(formService.getRenderedTaskForm(deptLeaderTask.getId())); // #3
    }
}
```

执行单元测试时需要把所有使用到的表单文件和流程定义一同部署，所以在代码清单6-19的#1处用一个字符串组作为部署参数，在#2和#3处分别读取了启动流程的表单和任务表单。值得注意的是动态表单读取的表单返回的是一个Form对象（StartForm数据和TaskForm数据），而外置表单默认用纯文本格式返回表单文件的内容。

6.3.3 自定义表单引擎

既然FormService接口的getRendered*Form()方法返回一个纯文本的内容，为什么接口的返回类型是Object而不是String呢？在第1章中就提到了Activiti是一个BPM平台，不管是B/S还是C/S结构的应用都可以使用Activiti作为流程引擎，对于B/S结构的应用来说，一般用HTML作为表现层，但是对于C/S来说HTML的纯文本代码明显不能作为C/S的基础组件，例如Java的Swing、SWT等通过Java对象的方式显示控件，那么如何生成适用于C/S程序的表单呢（可以由Java的Swing、AWT、SWT控件对象组成）？

Activiti支持自定义表单引擎以适应各种场景，默认的表单引擎（org.activiti.engine.impl.form.JuelFormEngine）是基于Juel实现的，该引擎可以计算表单文件中EL表达式作为表单内容，所以当调用FormService接口的getRendered*Form()方法后得到的内容是经过Form引擎处理过的。

继续回答如何让Activiti生成C/S程序所需的Java控件：创建自定义的表单引擎。和自定义表单字段类型类似，创建自定义表单引擎只需要实现接口org.activiti.engine.impl.form.FormEngine，然后像注册自定义表单类型一样在引擎中注册自定义的表单引擎实现类。

代码清单6-20列出了自定义表单引擎MyFormEngine的简单实现，可以生成一个Swing的按钮控件。

代码清单6-20 自定义表单引擎

```
public class MyFormEngine implements FormEngine {
    @Override
    public String getName() {
        return "myformengine"; // 表单引擎的名称
    }
    @Override // 生成启动流程表单
    public Object renderStartForm(StartFormData startForm) {
        javax.swing.JButton jButton = new javax.swing.JButton();
        jButton.setName("My Start Form Button");
        return jButton;
    }
    @Override // 生成用户任务表单
    public Object renderTaskForm(TaskFormData taskForm) {
        javax.swing.JButton jButton = new javax.swing.JButton();
        jButton.setName("My Task Form Button");
        return jButton;
    }
}
```

定义了表单引擎实现之后在引擎配置中进行注册配置，如代码清单6-21所示。

代码清单6-21 注册自定义的表单引擎

```
<bean id="processEngineConfiguration"
      class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">
    <property name="customFormEngines">
        <list>
            <bean class="me.kafeitu.activiti.chapter6.form.MyFormEngine" />
        </list>
    </property>
</bean>
```

接着用单元测试验证我们的表单引擎是否能够正常工作，代码清单6-22列出了测试自定义表单引擎的代码。

代码清单6-22 测试自定义表单引擎

```
public class MyFormEngineTest extends AbstractTest {
    @Test
    @Deployment(resources = { "chapter6/leave-formkey/leave-formkey.bpmn",
                           "chapter6/leave-formkey/leave-start.form" })
    public void allPass() throws Exception {
        ProcessDefinition processDefinition = repositoryService
            .createProcessDefinitionQuery().singleResult();
        // 读取启动表单
        Object renderedStartForm = formService
            .getRenderedStartForm(processDefinition.getId(), "myformengine"); #1
        // 验证表单对象
        assertNotNull(renderedStartForm); #2-S
        assertTrue(renderedStartForm instanceof javax.swing.JButton);
        javax.swing.JButton button = (JButton) renderedStartForm;
        assertEquals("My Start Form Button", button.getName()); #2-E
    }
}
```

在代码清单6-22中，#1处在调用接口获取表单内容时添加了一个参数“myformengine”，明确指定了生成表单不再使用默认的表单引擎，而是使用在引擎中注册过且名称为“myformengine”的表单引擎生成表单内容。#2处用几个断言验证我们的表单引擎是否可以创建我们所需要的基于Swing的按钮控件。

6.3.4 读取流程启动表单

在6.1.3节把一个动态表单的例子部署到了Activiti Explorer中，在启动流程时可以看到一个根据表单字段生成的表单，但是很遗憾Activiti Explorer不支持外置表单，所以本节基于对外置表单的理解为Activiti Explorer添加外置表单支持。

首先把图6-30中的压缩文件leave-formkey.zip部署到引擎中，完成部署后列表中多出了一条外置表单的请假流程，如图6-31所示。

流程定义ID	部署ID	流程定义名称	流程定义KEY	版本号	XML资源名称	图片资源名称	操作
leave-formkey:1:920	913	请假流程-外置表单	leave-formkey	1	leave-formkey.bpmn		 删除  启动
leave:1:204	201	请假流程-动态表单	leave	1	leave.bpmn	leave.png	 删除  启动
leave:2:912	909	请假流程-动态表单	leave	2	leave.bpmn	leave.png	 删除  启动

图6-31 部署了请假流程的外置表单流程定义

单击“启动”按钮后显示如图6-32的页面，表单内容为空。



图6-32 没有表单内容的流程启动界面

6.2.2节的读取表单的代码不能读取外置表单内容，这里利用6.3.1和6.3.2所学的技术要点把原来的方法稍加改造即可支持外置表单的读取。改造的方法很简单，可以通过流程定义对象ProcessDefinition（图6-31中列表的列即流程定义对象的属性）的hasStartFormKey()方法进行验证。代码清单6-23列出了如何判断是否有外置表单。

代码清单6-23 根据流程定义的方法判断是否有外置表单

```
public class ProcessDefinitionController extends AbstractController {
    @RequestMapping(value = "getform/start/{processDefinitionId}")
    public ModelAndView readStartForm(@PathVariable("processDefinitionId") String processDefinitionId) throws Exception {
        ProcessDefinition processDefinition=repositoryService.createProcessDefinitionQuery()
            .processDefinitionId(processDefinitionId).singleResult();
        boolean hasStartFormKey = processDefinition.hasStartFormKey(); #1
        // 根据是否有formKey属性判断使用哪个展示层
        String viewName = "chapter6/start-process-form";
        ModelAndView mav = new ModelAndView(viewName);
        if (hasStartFormKey) { // 判断是否有formKey属性 #2-S
            Object renderedStartForm = formService.getRenderedStartForm(processDefinitionId);
            mav.addObject("startFormData", renderedStartForm);
            mav.addObject("processDefinition", processDefinition);
        } else { // 动态表单字段
            StartFormData startFormData = formService.getStartFormData(processDefinitionId);
            mav.addObject("startFormData", startFormData);
        } #2-E
        mav.addObject("hasStartFormKey", hasStartFormKey);
        mav.addObject("processDefinitionId", processDefinitionId);
        return mav;
    }
}
```

代码清单6-23在代码清单6-7的基础上进行了改进，改进后可以支持读取动态表单和外置表单，主要改进有以下两点。

- #1处从流程定义对象中获取是否具有外置表单的特征（formKey属性不为空）。
 - #2处根据变量hasStartFormKey的值决定使用哪个方法读取表单，当值为true时使用外置表单的方式读取，否则使用动态表单的方式读取。
-  **说明** 外置表单文件的内容就是普通的HTML，读者可以打开本书配套资源文件的chapter6-oa-forms/src/main/resources/chapter6/leave-formkey目录中的form文件查看。
- 如此，单击图6-31中第一条记录的“启动”按钮将显示图6-33所示的页面，从图中可以看出表单内容和leave-start.form文件的内容一致。

启动流程— [请假流程-外置表单]，版本号：1

开始时间：

结束时间：

请假原因：

◀ 返回列表

▶ 启动流程

图6-33 请假流程——外置表单启动界面

要兼容动态表单和外置表单表现层也需要做出调整，根据不同的情况处理表单内容。代码清单6-24列出了基于代码清单6-8调整后的start-process-form.jsp文件的代码片段。

代码清单6-24 基于代码清单6-8调整后的start-process-form.jsp文件的代码片段

```
<h3>启动流程—
    <c:if test="${hasStartFormKey}">
        [{processDefinition.name}], 版本号: ${processDefinition.version}
    </c:if>
    <c:if test="${!hasStartFormKey}">
        [{startFormData.processDefinition.name}], 版本号:
        ${startFormData.processDefinition.version}
    </c:if>
</h3>
<form action="${ctx }/chapter6/process-instance/start/${processDefinitionId}"
    class="form-horizontal" method="post">
    <c:if test="${hasStartFormKey}">
        ${startFormData}
    </c:if>
    <c:if test="${!hasStartFormKey}">
        <c:forEach items="${startFormData.formProperties}" var="fp">
            ... // 此处省略了代码清单6-8的html元素转换
        </c:if>
    </c:if>
</form>
```

代码清单6-24主要是根据hasStartFormKey的值进行判断处理，如果是外置表单，直接将字符型的变量startFormData输出即可，因为表单内容是纯文本的HTML代码。

最后单击图6-32的“启动流程”按钮向后台提交启动流程请求，同样需要做出调整以兼容外置表单。需要调整的是读取表单字段值的部分，因为动态表单可以通过读取流程定义的启动表单字段列表得知需要从请求参数列表读取哪些表单字段值。对于外置表单，因为没有在流程定义文件中进行描述，只是把表单内容都保存在单独的form文件中，所以对于这样的情况只能读取所有的请求参数作为流程启动参数。代码清单6-25列出了兼容读取动态表单和外置表单内容的代码。

代码清单6-25 启动流程时根据表单类型使用不同的表单参数处理方式

```
public String startProcessInstance(@PathVariable("processDefinitionId") String pdid, HttpServletRequest request) {
    ProcessDefinition processDefinition = repositoryService.createProcessDefinitionQuery()
        .processDefinitionId(pdid).singleResult();
    boolean hasStartFormKey = processDefinition.hasStartFormKey(); // #1
    Map<String, String> formValues = new HashMap<String, String>();
    if (hasStartFormKey) { // 外置表单
        Map<String, String[]> parameterMap = request.getParameterMap(); // #2
        Set<Entry<String, String[]>> entrySet = parameterMap.entrySet();
        for (Entry<String, String[]> entry : entrySet) {
            String key = entry.getKey();
            formValues.put(key, entry.getValue()[0]);
        }
    } else { // 动态表单
        // 先读取表单字段，再根据表单字段的ID读取请求参数值
        StartFormData formData = formService.getStartFormData(pdid);
        // 从请求中获取表单字段的值
        List<FormProperty> formProperties = formData.getFormProperties();
        for (FormProperty formProperty : formProperties) {
            String value = request.getParameter(formProperty.getId());
            formValues.put(formProperty.getId(), value);
        }
    }
}
```

在代码清单6-25的#1处获取是否有formKey属性。在#2处处理有formKey属性的情况，读取全部请求参数列表，动态表单的处理方式保持不变。

代码清单6-25读取了请求参数集中的所有参数作为启动流程参数，如果有写字段值不需要保存到数据库，就会导致引擎数据库多出很多垃圾数据。在实际应用中可以设定一些规则处理，比如需要保存到数据库的字段以一个特殊的前缀开头。

```
<input type="text" id="endTime" name="fp_endTime"/>
```

以一个特殊的前缀“fp_”开头，表示这个字段需要保存到数据库，这样在处理请求参数时就可以根据这个特殊的前缀进行过滤，伪代码如下：

```
if (parameter.startsWith("fp_")) {
    String[] paramSplit = parameter.split("_");
    formProperties.put(paramSplit[1], entry.getValue()[0]);
}
```

单击图6-32中的“启动流程”按钮后成功启动一个流程实例，在图6-34中列出了包含“请假流程-外置表单”流程任务ID为1511的任务。

任务ID	任务名称	流程实例ID	流程定义ID	任务创建时间	办理人	操作
1511	部门经理审批	1501	leave-formkey:1:1445	Sun Dec 02 22:44:42 CST 2012		

图6-34 请假流程——外置表单的任务

 **说明** 使用activiti:formKey属性除了可以指定一个form文件的名称外，还可以指定一个页面的访问路径，在jBPM4中使用过这种方法的读者可以继续使用，通过FormService的接口读取formKey属性后再访问表单内容。

6.3.5 任务签收与办理

单击图6-34的“签收”按钮并单击之后的“办理”按钮跳转到任务办理页面，如图6-35所示。

任务办理—[部门经理审批]，流程定义ID: [leave-formkey:1:1445]

申请人:	kermit
开始时间:	startDate
结束时间:	2012-12-02
请假原因:	公休
审批意见:	同意
返回列表 完成任务	

图6-35 任务办理——读取外置表单

从图6-35中可以看到流程定义ID为“leave-formkey:1:1445”，表明该任务是“请假流程-外置表单”流程定义产生的。和读取启动流程表单兼容动态表单和外置表单一样，对于任务表单同样需要兼容两种不同的表单形式。

代码清单6-26在代码清单6-12的基础上添加了一个变量hasFormKey，以便在展示层可以作为判断表单类型的依据。

代码清单6-26 基于代码清单6-12添加hasFormKey变量

```
@RequestMapping(value = "task/getform/{taskId}")
public ModelAndView readTaskForm(@PathVariable("taskId") String taskId) throws Exception {
    String viewName = "chapter6/task-form";
    ModelAndView mav = new ModelAndView(viewName);
    TaskFormData taskFormData = formService.getTaskFormData(taskId);
    if (taskFormData.getFormKey() != null) { // formKey不为空表示是外置表单
        Object renderedTaskForm = formService.getRenderedTaskForm(taskId);
        Task task = taskService.createTaskQuery().taskId(taskId).singleResult();
        mav.addObject("task", task);
        mav.addObject("taskFormData", renderedTaskForm);
        mav.addObject("hasFormKey", true); // 标记为外置表单
    } else {
        mav.addObject("taskFormData", taskFormData);
    }
    return mav;
}
```

对应的展示层也需要进行调整，类似代码清单6-24，代码清单6-27列出了任务办理页面的调整。

代码清单6-27 调整任务办理页面以兼容外置表单

```
<h3>任务办理-[${hasFormKey ? task.name : taskFormData.task.name}], 流程定义ID: [${hasFormKey ? task.processDefinitionId : taskFormData.task.processDefinitionId}]</h3>
<form action="${ctx }/chapter6/task/complete/${hasFormKey ? task.id : taskFormData.task.id}" class="form-horizontal" method="post">
    <c:if test="${hasFormKey}">${taskFormData}</c:if> // 直接输出外置表单内容
    <c:if test="${!hasFormKey}">
        <c:forEach items="${taskFormData.formProperties}" var="fp">
            http://www.hzcourse.com/resource/readBook?path=/openresources/teach_ebook/uncompressed/15030/OEBPS/Text/... // 省略了部分代码
        </c:forEach>
    </c:if>
</form>
```

代码清单6-27的加粗部分就是用hasFormKey变量进行的判断处理，可以很好地兼容两种表单的内容显示。不过从Activiti 5.16版本开始可以不借助hasFormKey变量判断表单类型，因为在此版本中可以从Task对象中获取到formKey属性，该属性值也会保存在运行时和历史任务表的“FORM_KEY_”字段中。

外置表单除了配置一个文件的相对路径外还可以配置URL，例如把Activiti作为流程中心（基于Activiti Rest服务）对外提供服务时可以使用这种方式分离业务与流程，也就是Activiti负责业务流程的驱动，而业务的表单还是交给各个业务系统处理，相关内容读者可以参考20.7.2节。

6.4 本章小结

本章内容讲解了Activiti支持的两类表单：动态表单、外置表单，这是流程引擎系统中最重要的一部分内容，是机器与人类交互的重要载体。针对每一种表单都进行了以下几方面的详细讲解：

- 如何在流程定义中定义表单信息（表单字段、表单名称）。
- 如何在启动流程时读取表单。
- 如何读取任务的表单内容。
- 如何保存表单字段内容到引擎数据库。

在动态表单部分还对如何自定义表单字段类型进行了详细的讲解，以此满足对于现有控件或特殊字段类型的需求。

在外置表单中用实例方式讲解如何定义自己的动态表单引擎，使外置表单的使用方位不再局限B/S架构，通过自定义表单引擎同样可以利用Activiti引擎创建适用于C/S架构的表单对象。

第7章 Activiti与容器集成

Spring是由Rod Johnson发起的轻量级开发框架，经过多年的发展已经非常成熟，并且应用于大多数企业应用中，通过简单的配置即可和其他的框架集成，利用IOC实现解耦合，利用AOP拦截事务、日志等。

Activiti从最开始设计时就考虑了与Spring的集成，从引擎的默认配置文件（activiti.cfg.xml）就可以看出，其实Activiti的配置文件解析及XML规范都是基于Spring扩展的。这一特点对于Activiti和现有系统的集成可以说是水到渠成，所以本章的内容将从各个方面讲解Activiti的Spring模块（activiti-spring）可以帮我们做哪些事情；如何和现有系统进行集成，包括Bean的注入、统一事务管理等。

7.1 流程引擎工厂

通过JDBC方式访问数据库首先需要加载对应数据库的驱动，只有基于JDBC标准实现的数据库驱动才能和目标数据库通信。在流程引擎中创建引擎是第一步，只有通过引擎对象才能操作引擎数据库（通过接口调用）。

创建引擎对象的方式有很多种，在下面的章节中我们会学到使用多种方式创建引擎的方法，例如创建基于内存数据库的引擎用于单元测试，或者通过引擎配置对象动态配置后创建引擎对象。

7.1.1 创建引擎的方式

下面的代码可以通过ProcessEngineConfiguration类创建一个动态的基于内存数据库的引擎对象。

```
ProcessEngine processEngine = ProcessEngineConfiguration
    .createStandaloneInMemProcessEngineConfiguration()
    .buildProcessEngine();
```

在单元测试中还可以使用ActivitiRule类创建用于测试的引擎对象，例如下面的代码均可创建一个用于测试的引擎对象。

```
@Rule
ActivitiRule activitiRule = new ActivitiRule();
ProcessEngine engine = activitiRule.getProcessEngine();
// 指定一个配置文件名称
String resourceName = "activiti.cfg.xml";
@Rule
ActivitiRule activitiRule = new ActivitiRule(resourceName);
ProcessEngine engine2 = activitiRule.getProcessEngine();
```

当需要多个引擎时还可以通过设置引擎的名称区分其他的引擎对象，例如下面的代码创建了一个名称为“oracleEngine”的引擎，并且动态设置了数据库参数，最后通过ProcessEngines类读取名称为“oracleEngine”的引擎对象。

```
ProcessEngineConfiguration.createProcessEngineConfigurationFromResourceDefault()
    .setProcessEngineName("oracleEngine")
    .setJdbcDriver("oracle.jdbc.Driver")
    .setJdbcUrl("jdbc:oracle:thin:@localhost:1521:XE")
    .setJdbcUsername("dbusername")
    .setJdbcPassword("password")
    .setDatabaseType("oracle")
    .buildProcessEngine();
ProcessEngine oracleEngine =
    ProcessEngines.getProcessEngine("oracleEngine");
```

当创建流程引擎时，每个引擎都有一个唯一的名称；当创建引擎对象时，为显示地调用setProcessEngineName()方法，引擎会使用默认的引擎名称“default”作为引擎名称，所以下面代码的执行结果为true。

```
ProcessEngine processEngine = ProcessEngineConfiguration
    .createStandaloneInMemProcessEngineConfiguration()
    .buildProcessEngine();
assertEquals("default", processEngine.getName());
```

7.1.2 配置ProcessEngineFactoryBean

与Spring集成的目的是把获取引擎和服务的方式由Spring代理统一管理，以及业务与流程引擎的统一事务管理（这也是与Spring集成的一个重要原因）。Activiti提供了一个专门的模块“activiti-spring”，由Spring团队的布道者Josh Long主导开发。

在Maven项目中使用Activiti的Spring模块，需要在pom.xml文件中添加activitispring模块的依赖，依赖代码如下：

```
<dependency>
    <groupId>org.activiti</groupId>
    <artifactId>activiti-spring</artifactId>
    <version>5.11</version>
</dependency>
```

在传统项目中，需要把activiti-spring-5.x.jar引入到classpath中（Web类型的项目放在/WEB-INF/lib目录下）。

不管采用哪种引擎创建方式，都先设置一系列参数，然后根据这些参数创建引擎对象。对于Spring方式也是一样，首先需要配置一个类名为“org.activiti.spring.SpringProcessEngineConfiguration”的Bean对象来设置创建引擎所需的参数。代码清单7-1列出了基于Spring的引擎配置Bean对象的XML定义片段。

代码清单7-1 基于Spring的引擎配置Bean对象的XML定义片段

```
<bean id="processEngineConfiguration"
    class="org.activiti.spring.SpringProcessEngineConfiguration"> #1
```

```
...
</bean>
<bean id="processEngine" class="org.activiti.spring.ProcessEngineFactoryBean">    #2
    <property name="processEngineConfiguration" ref="processEngineConfiguration" />
</bean>
```

代码清单7-1简单描述了基于Spring的引擎创建方式，首先在#1处创建一个引擎配置Bean，其id为“processEngineConfiguration”，当然这个id可以随意定义，但是要保证其唯一性。

在#2处把引擎配置对象注入类名为“org.activiti.spring.ProcessEngineFactoryBean”的Bean中。

如此，当加载Spring配置文件创建上下文对象（ApplicationContext）时就会创建定义的两个Bean对象。

在SpringProcessEngineConfiguration中的配置和在activiti.cfg.xml中的配置基本类似，可以配置数据库参数或引擎相关的参数。但是基于Spring创建引擎的方式有一点需要特别声明，那就是事务管理器。

对于7.1.1节列举的创建引擎的方式，引擎采用Mybatis内置的事务管理器，如果使用Spring方式创建引擎，就必须明确指定一个事务管理器。

代码清单7-2列出了基于Spring的引擎配置以及代理Service的XML配置的代码，XML文件位于chapter7-spring/src/test/resources/applicationContext-test.xml。

代码清单7-2 基于Spring的引擎配置以及代理Service的XML配置

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" xmlns:context="http://www.springframework.org/schema/context"
xmlns:tx="http://www.springframework.org/schema/tx" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" // 引入xsd文件
xsi:schemaLocation="http://www.springframework.org/schema/beans // 以及xmlns声明
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.1.xsd">
    <bean id="dataSource" // 创建基于内存数据库的数据源 #1-S
        class="org.springframework.jdbc.datasource.SimpleDriverDataSource">
        <property name="driverClass" value="org.h2.Driver" />
        <property name="url" value="jdbc:h2:mem:activiti;DB_CLOSE_DELAY=1000" />
        <property name="username" value="sa" />
        <property name="password" value="" />
    </bean>
    <bean id="transactionManager" // 定义事务管理器，并注入数据源 #2-S
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource" />
    </bean>
    <tx:annotation-driven transaction-manager="transactionManager" // 注解方式配置事务
        proxy-target-class="true" /> #2-E
    <bean id="processEngineConfiguration" // 定义基于Spring的引擎配置对象Bean #3-S
        class="org.activiti.spring.SpringProcessEngineConfiguration">
        <property name="dataSource" ref="dataSource" />
        <property name="transactionManager" ref="transactionManager" />
        <property name="databaseSchemaUpdate" value="true" />
        <property name="jobExecutorActivate" value="false" />
    </bean> #3-E
    // 配置引擎工厂Bean
    <bean id="processEngineFactory" #4-S
        class="org.activiti.spring.ProcessEngineFactoryBean">
        <property name="processEngineConfiguration" ref="processEngineConfiguration" />
    </bean> #4-E
    <bean id="repositoryService" factory-bean="processEngineFactory" #5-S
        factory-method="getRepositoryService" />
    <bean id="runtimeService" factory-bean="processEngineFactory"
        factory-method="getRuntimeService" />
    <bean id="formService" factory-bean="processEngineFactory"
        factory-method="getFormService" />
    <bean id="identityService" factory-bean="processEngineFactory"
        factory-method="getIdentityService" />
    <bean id="taskService" factory-bean="processEngineFactory"
        factory-method="getTaskService" />
    <bean id="historyService" factory-bean="processEngineFactory"
        factory-method="getHistoryService" />
    <bean id="managementService" factory-bean="processEngineFactory"
        factory-method="getManagementService" /> #5-E
</beans>
```

代码清单7-2的配置分为几个部分：XML的命名空间、数据源、事务管理器、引擎配置、引擎工厂、获取Activiti的Service Bean。

xml头部声明了几个XML的命名空间，相比activiti.cfg.xml多出了前缀为tx的命名空间，用来定义事务方面的配置。

#1处使用Spring的简单数据源类创建了数据源，使用的是内存数据库H2，和内存引擎实现类StandaloneInMemProcessEngineConfiguration的配置类似。

#2处定义了事务管理器Bean对象，并用tx.annotation-driven声明使用注解配置事务。注解事务是可选的，读者可以继续使用XML方式声明事务管理机制。

#3处在代码清单7-1已经介绍过了，这里在Spring引擎配置对象中注入了几个必要的属性，尤其是事务管理器是必不可少的。

#4处定义一个引擎工厂，并注入了引擎配置Bean对象processEngineConfiguration，如此通过工厂对象就可以创建一个引擎对象（引擎对象是受保护类型）。

#5处分别用Spring提供的工厂方式获取Activiti的7个Service接口对象，这样就可以很方便地使用Spring的工具类获取Service实例或者通过注解方式将Activiti的Service接口对象注入被Spring代理的Bean中。

代码清单7-3针对代码清单7-2的配置进行了测试，以验证其能够正确创建引擎并获取Activiti的Service实例。

代码清单7-3 测试代码清单7-2的配置是否可以创建引擎对象

```
public class CreateEngineUseSpringProxy {
    @Test
    public void createEngineUseSpring() {
        ClassPathXmlApplicationContext context = new
            ClassPathXmlApplicationContext("applicationContext-test.xml");
        ProcessEngineFactoryBean factoryBean =
            context.getBean(ProcessEngineFactoryBean.class);
        assertNotNull(factoryBean); // 验证获取的工厂对象是否不为空
        RuntimeService bean = context.getBean(RuntimeService.class);
        assertNotNull(bean); // 验证能否正常获取RuntimeService对象
    }
}
```

代码清单7-3通过创建ClassPathXmlApplicationContext对象的方式解析Spring配置文件读取被代理对象，运行单元测试后顺利通过，如图7-1所示。

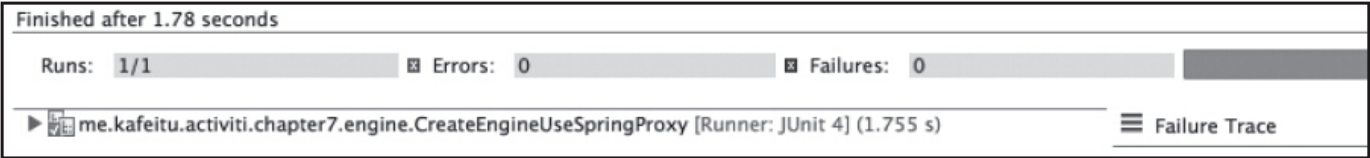


图 7-1 运行代码清单7-3的测试代码的结果

除了使用传统的方式获取Bean对象之外，还可以使用基于注解的方式，如代码清单7-4所示。

代码清单7-4 使用注解方式测试代码清单7-2的配置是否可以创建引擎对象

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext-test.xml")
public class CreateEngineUseSpringProxyByAnnotation {
    @Autowired
    RuntimeService runtimeService; // 注入RuntimeService #1
    @Autowired
    ProcessEngineFactoryBean factoryBean; // 注入工厂类 #2
    @Test
    public void testService() throws Exception {
        assertNotNull(runtimeService); #3
        ProcessEngine processEngine = factoryBean.getObject(); #4
        assertNotNull(processEngine.getRuntimeService()); #5
    }
}
```

代码清单7-4使用注解方式由Spring注入Bean对象，在#1处注入了RuntimeService并在#3处验证注入的runtimeService变量不为空；在#4处通过代码调用方式获取引擎对象，这也是在代码清单7-2#5处的Java代码实现。

7.2 自动部署流程定义

自动部署是通过Spring方式创建流程引擎的一个独有功能，可以在初始化引擎时自动把定义的资源部署到引擎中。部署的方式是利用Spring提供的Resource^[1]功能实现的，可以使用多种方式配置资源路径，例如classpath、file等。

但是关于这个功能，有些读者可能会问，如果每次启动应用都会加载Spring配置文件并初始化流程引擎，岂不是每次都部署一个新的版本？

答案是不会，在5.2.1节中介绍了使用DeploymentBuilder部署流程定义，细心的读者可能发现了一个方法为enableDuplicateFiltering()，这个方法的作用是排除重复的流程定义，所以只有流程数据库中还没有和自动部署的流程定义相同的记录才会部署，否则会忽略，不会出现启动多次应用后部署多个版本的流程定义问题。

有了理论基础后再通过单元测试方式学习如何配置自动部署流程定义，代码清单7-5在代码清单7-2的基础上添加了自动部署资源的配置项，文件名为applicationContext-autodeployment.xml。

代码清单7-5 添加了自动部署资源的引擎配置

```
<bean id="processEngineConfiguration" class="org.activiti.spring.SpringProcessEngineConfiguration">
    <property name="dataSource" ref="dataSource" />
    <property name="transactionManager" ref="transactionManager" />
    <property name="databaseSchemaUpdate" value="true" />
    <property name="jobExecutorActivate" value="false" />
    <property name="deploymentResources"
        value="classpath*:chapter6/leave-formkey/leave-formkey.bpmn"></property>
</bean>
```

代码清单7-5中添加的deploymentResources属性配置了一个classpath方式的资源路径，读取classpath中的leave-formkey.bpmn文件，如果有多个资源，还可以使用通配符形式匹配多个资源文件，例如classpath*:chapter6/**/*.bpmn可以匹配到所有chapter6目录下（包括子目录）扩展名为bpmn的文件。

对应的单元测试代码见代码清单7-6所示。

代码清单7-6 测试流程定义自动部署

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext-autodeployment.xml")
public class AutoDeploymentUseSpring {
    @Autowired
    RepositoryService repositoryService;
    @Test
    public void testAutoDeployment() {
        long count = repositoryService.createProcessDefinitionQuery().count();
        assertEquals(1, count);
    }
}
```

代码清单7-6的代码比较简单，仅仅通过流程定义查询接口统计已部署的流程定义数量，期望结果为1表示和预期一致。

如果代码清单7-5中的deploymentResources配置的是 "classpath*:chapter6/**/*.bpmn"，那么查询到的流程定义数量应该为2。

此功能对于新系统的上线或者在开发过程中加入新的流程非常有用，可以自动部署引擎数据库中不存在的或者修改过的流程定义。

[1] <http://static.springsource.org/spring/docs/3.1.3.RELEASE/spring-framework-reference/html/resources.html>。

7.3 表达式

7.3.1 表达式基础

我们知道，在流程定义中几乎所有的属性都可以使用变量方式定义，这样可以在运行中灵活地设置，例如在第6章请假流程的流程定义文件中，对于领导与人事的审批判断使用变量来判断执行排他分支的哪一条输出流。

流程定义文件中出现的判断条件的语句或者调用一个Bean的方法都称为表达式。对表达式的解析由JUEL根据UEL (Unified Expression Language) 规范实现，之后Activiti又对JUEL^[1]进行了扩展以满足更多的需求。

相信熟悉JSP的读者对UEL并不陌生了，通过`{}`包裹的表达式即可读取变量值、计算变量值等操作。不过，经过Activiti改造过的UEL功能更加的强大，表7-1列举了一些表达式的典型应用。

表7-1 UEL表达式的应用示例

表 达 式	表达式含义
<code>\${myVar}</code>	获取变量名称为 myVar 的变量值
<code>\${myVar.name}</code>	从变量 myVar 中读取 name 属性
<code>\${serviceBean.confirm()}</code>	执行变量名称为 serviceBean 的 confirm() 方法
<code>\${serviceBean.confirm('henryyan')}</code>	执行变量名称为 serviceBean 的 confirm() 方法，同时以字符串 henryyan 作为参数。这里字符串也可以使用变量代替，如果在变量中包含一个名称为 name 的变量，就可以动态设置方法的参数，例如 <code>\${serviceBean.confirm(name)}</code>
<code>\${serviceBean.confirm(name, execution)}</code>	执行变量名称为 serviceBean 的 confirm() 方法，并且传入名称为 name 的变量作为第一个参数，传入引擎内置的变量 execution 作为第二个参数。关于引擎内置的变量见表 7-2

表7-2 引擎内置的三个变量

变 量 名 称	变 量 含 义
execution	此变量在运行阶段总是可以调用，对应接口：<code>org.activiti.engine.delegate.DelegateExecution</code>，可以获取流程实例的变量，包含了一些执行期的信息项，例如流程实例 ID、业务 ID（<code>businessKey</code>，如果在启动流程时指定了就可以获取）、当前节点等信息，另外在 5.11 版本中还可以直接获取引擎的 7 个 Service 接口 如果在流程定义中添加了执行监听器（实现 <code>ExecutionListener</code> 接口）或者为某个活动添加了 <code>Delegate</code>（实现 <code>JavaDelegate</code> 接口），那么接口将提供一个 <code>DelegateExecution</code> 的对象作为参数

(续)

变 量 名 称	变 量 含 义
task	相对于 <code>execution</code> 来说 <code>task</code> 变量的作用域就比较小了，<code>execution</code> 可以在整个运行期的所有活动上使用，但是 <code>task</code>（<code>DelegateTask</code> 接口）仅仅支持用户任务（<code>UserTask</code>），而且限定为 <code>expression</code> 类型的表达式。所以通过此变量可以设置或者读取任务相关的变量，设置任务办理人或者候选人等，基本涵盖了用户任务相关的所有属性以及特性。 例如，下面的配置可以在一个用户任务创建时执行一个表达式，并在执行时把 <code>DelegateTask</code> 的对象传给被调用对象的方法： <pre><userTask id="usertask1" name="在用户任务上添加表达式"> <extensionElements> <activiti:taskListener event="create" expression="\${myBean.invokeTask(task)}"> </activiti:taskListener> </extensionElements> </userTask></pre>
authenticatedUserId	此变量仅在启动流程实例前调用 <code>IdentityService</code> 的 <code>setAuthenticatedUserId()</code> 方法时才会由引擎提供，获取的方式很简单，执行 <code>\${authenticatedUserId}</code> 即可

在Activiti中表达式应用非常广泛，通过表达式可以动态计算（读取）一切可以计算的表达式，从而获取动态值。例如可以动态设置用户任务的办理人（`activiti:assignee`属性），执行一个Java Service 任务，或者执行`Execution`监听器、`Task`监听器，还有在请假流程中排他分支的输出流条件判断等。

在Activiti中，所有表达式中出现的变量均需要实现序列化接口java.io.Serializable。在前面的讲解中通过各种接口设置的变量均保存到了数据库，类型有字符串、日期、数字等，这些基本格式类在JDK中均已实现序列化接口。对于自定义的JavaBean来说，Activiti可以保存一个自定义的Bean对象到数据库。

如果一个自定义Bean没有实现序列化接口，那么将会得到类似如下的异常提示：

```
org.activiti.engine.ActivitiException: couldn't find a variable type that is able to serialize me.kafeitu.activiti.chapter7.expression.MyBean@58bd3b2d
```

对于表7-2中的三个内置变量，需要特别说明一下，仅能通过表达式获取，不能通过RuntimeService或TaskService的getVariable()方法获取。

7.3.2 表达式示例

程序员在解决一个问题时最喜欢看到的是具体的代码，有了前面的理论的基础，下面就用单元测试的方式帮助彻底理解表7-1和表7-2中表达式及内置变量如何应用。

首先设计了一个用于测试的流程定义，其ID为“expression”，该流程定义包含三个节点，分别用于测试不同的表达式，如图7-2所示。

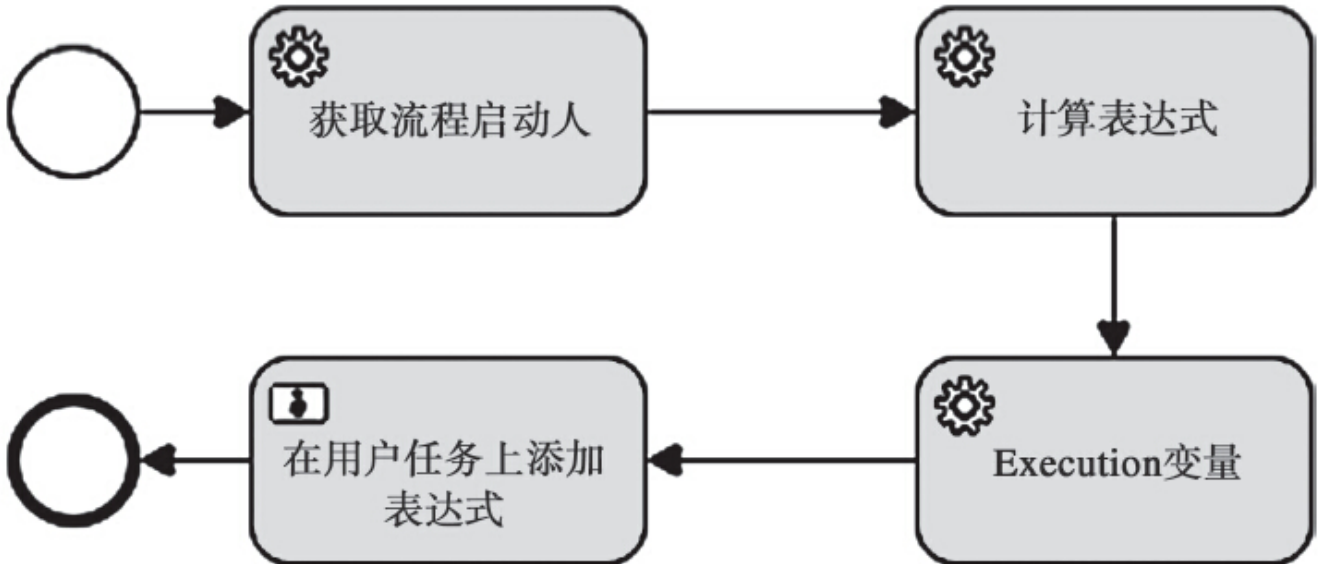


图7-2 理解表达式的辅助流程定义

图7-2对应的流程定义XML文件见代码清单7-7。

代码清单7-7 图7-2所示的流程定义XML

```
<process id="expression" name="expression">
  <startEvent id="startevent1" name="Start"></startEvent>
  <serviceTask id="execExpression" name="计算表达式" #1-S
    activiti:expression="${myBean.print(name)}" // 执行myBean的print()方法
    activiti:resultVariableName="returnValue" // 结果保存到变量returnValue中
  </serviceTask> #1-E
  <userTask id="usertask1" name="在用户任务上添加表达式"> #2-S
    <extensionElements> // 调用myBean的invokeTask方法，并传递内置变量task
      <activiti:taskListener event="create" expression="${myBean.invokeTask(task)}" /> #2-1
    </extensionElements>
  </userTask> #2-E
  <serviceTask id="getAuthenticatedUserId" name="获取流程启动人" #3-S
    activiti:expression="${authenticatedUserId}" // 获取内置变量authenticatedUserId
    activiti:resultVariableName="authenticatedUserIdForTest"></serviceTask> #3-E
  <serviceTask id="executionExample" name="Execution变量" #4-S
    activiti:expression="${myBean.printByKey(execution)}"
    activiti:resultVariableName="businessKey"></serviceTask> #4-E
</process>
```

代码清单7-7基本涵盖了表达式的几种使用方式，#1处演示了调用一个Bean对象的方法并将一个名称为name的自定义变量作为方法的参数；#2处在#1的基础上更改了参数，将引擎内置的变量task作为被调用方法的参数；#3处获取引擎内置的变量authenticatedUserId，并演示了如何把表达式的结果保存到另外一个变量中，如在#3-E处把authenticatedUserId的值保存在名称为authenticatedUserIdForTest的变量中；#4处则融合了#2处和#3处的特性，既在调用方法时传入引擎内置变量execution，又把方法的执行结果保存到变量businessKey中。

如果对于本节开头的文字内容理解得差不多，再加上代码清单7-7的示例，相信读者对于表达式的应用已经掌握得差不多了，下面对这一切的“假设”用代码来——验证。代码清单7-8对代码清单7-7做了完整的测试。

代码清单7-8 代码清单7-7对应的单元测试代码

```
public class ExpressionTest extends AbstractTest {
  @Test
  @Deployment(resources = "diagrams/chapter7/expression.bpmn")
  public void testExpression() {
    MyBean myBean = new MyBean(); // 首先将需要的变量初始化
    Map<String, Object> variables = new HashMap<String, Object>();
    variables.put("myBean", myBean); // 创建一个可以序列化的MyBean对象
    String name = "Henry Yan"; variables.put("name", name);
    // 运行期表达式
    identityService.setAuthenticatedUserId("henryyan"); // 设置启动流程人ID #1
    String businessKey = "9999"; // 业务ID
    ProcessInstance processInstance = runtimeService // 在启动流程时设置业务ID
      .startProcessInstanceByKey("expression", businessKey, variables); #2-S
    assertEquals("henryyan", runtimeService // 对应代码清单7-7的#3处
      .getVariable(processInstance.getId(), "authenticatedUserIdForTest")); #2-E
    assertEquals("Henry Yan, added by print(String name)", // 对应代码清单7-7的#1处 #3
      runtimeService.getVariable(processInstance.getId(), "returnValue"));
    assertEquals(businessKey, // 对应代码清单7-7的#4处
      runtimeService.getVariable(processInstance.getId(), "businessKey"));
    // DelegateTask 设置的变量
    Task task = taskService.createTaskQuery() // 查询任务并验证变量
      .singleResult();
    String setByTask = (String) taskService.getVariable(task.getId(), "setByTask");
  }
}
```



```
    assertEquals("I'm setted by DelegateTask, " + name, setByTask);    #4-E
}
// 自定义的Bean对象
class MyBean implements java.io.Serializable {    // 必须实现序列化接口    #5
    public void print() {    System.out.println("print content by print()");    }
    public String print(String name) {
        System.out.println("print content by print(String name), value is :"+ name);
        return name += " ", added by print(String name)";
    }
    public void invokeTask(DelegateTask task) {    // 接收名称为task的引擎内置变量
        task.setVariable("setByTask", "I'm setted by DelegateTask, " + task.getVariable("name"));
    }
    public String printBkey(DelegateExecution execution) {    // 接收名称为execution的引擎内置变量
        String processBusinessKey = execution.getProcessBusinessKey();
        System.out.println("process instance id: " + execution.getProcessInstanceId()
            + ", business key: " + processBusinessKey);
        return processBusinessKey;
    }
}
```

在代码清单7-8中，关键位置都已用文字说明，结合代码清单7-7的解释不难理解表达式的执行方式及过程。

从#5处开始一直到结束定义了一个名称为MyBean的类，并且实现了序列化接口，如此在#1处创建了一个MyBean类的对象并以myBean变量名称作为启动流程的参数（启动流程之后引擎会把myBean以二进制流的格式保存到数据库）。

针对#2处要特别说明一点，相比之前的情况这里在启动流程时添加了一个参数businessKey作为业务ID，当业务数据不保存在引擎数据库时，这个参数必须传入，否则无从得知流程实例属于哪个业务，使用外置表单时可以选择是否设置业务ID，一般在业务数据库保存成功后将得到的记录主键作为流程实例的业务ID，如此业务数据和流程数据就建立了关系。

#2-E、#3、#4处逐一验证流程执行的结果，其中#3和#4处的变量均被MyBean类的方法处理后返回。

7.3.3 使用Spring管理变量

使用Spring管理变量可以避免由引擎保存复杂变量（自定义的Bean）到数据库，如果变量需要事务管理Spring可以统一事务管理。

在代码清单7-8执行完成后，检索数据库的历史信息（对应表名ACT_HI_DETAIL），会得到一条如下的记录（在代码清单7-8的测试类中添加了输出数据库状态的代码，读者可自行查看）：

```
REV_ = 0
NAME_ = myBean
ACT_INST_ID_ = 6
VAR_TYPE_ = serializable
BYTEARRAY_ID_ = 12
ID_ = 13
TIME_ = 2012-12-14 22:19:33.537
TYPE_ = VariableUpdate
EXECUTION_ID_ = 5
PROC_INST_ID_ = 5
```

从上面的输出结果中可以判定，引擎把名称为myBean的对象持久化到了数据库，这也是为什么变量都需要实现序列化接口的原因。在输出结果中，字段“BYTEARRAY_ID_”对应的值为“12”，对应表ACT_GE_BYTEARRAY（此表保存了二进制流数据）中的记录。

本节将在7.3.2的基础上改造，由Spring代理复杂对象MyBean，而不再需要由引擎持久化到数据库。

改造的过程很简单，在代码清单7-8的开始我们创建了一个MyBean的实例，代码清单7-9把MyBean交给Spring代理，并且注入引擎中，如此，当获取编码myBena时就可以直接从Spring上下文中获取。

代码清单7-9 在Spring引擎实现中注入自定义Bean

```
<bean id="processEngineConfiguration" class="org.activiti.spring.SpringProcessEngineConfiguration"> #1
...
    <property name="beans"> #2-S
        <map> // entry的key属性就是变量名称
            <entry key="myBean" value-ref="myBean"></entry>
        </map>
    </property>
</bean> #2-E
<bean id="myBean" class="me.kafeitu.activiti.chapter7.expression.MyBean" /> #3
```

代码清单7-9的#2处通过设置引擎的预设参数“beans”设置了一个map集合，使用#3处id为“myBean”的Bean对象作为集合的元素，简单两步即完成了自定义Bean的注入。

当然除了使用#3处的XML定义之外，还可以使用注解方式由Spring自动扫描并代理，#2处的配置不变。

代码清单7-10的ExpressionWithSpringTest类继承了activiti-spring模块的一个测试基类SpringActivitiTestCase，和代码清单7-8的区别在于不再创建MyBean对象。

代码清单7-10 使用Spring代理自定义Bean作为流程变量的单元测试

```
@ContextConfiguration("classpath:applicationContext-expression.xml")
public class ExpressionWithSpringTest extends SpringActivitiTestCase {
    @Deployment(resources = {"diagrams/chapter7/expression.bpmn"})
    public void testExpression() {
        Map<String, Object> variables = new HashMap<String, Object>();
        String name = "Henry Yan";
        variables.put("name", name);
        // 其他代码和代码清单7-8一样，故省略
    }
}
```

再次运行单元测试，顺利通过后的结果如图7-3所示。

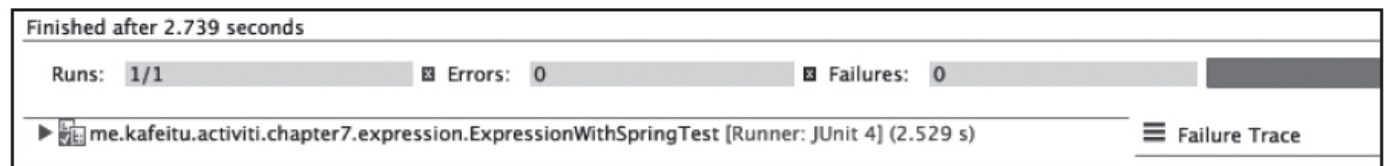


图7-3 代码清单7-10执行单元测试顺利通过

既然Activiti紧密结合Spring作为基础组件，能否不使用Spring也可以代理Bean呢？当然可以，代码清单7-11展示了在activiti.cfg.xml中添加自定义Bean的配置。

代码清单7-11 在Standalone引擎中注入自定义Bean

```
<bean id="processEngineConfiguration" class="org.activiti.engine.impl.cfg.StandaloneInMemProcessEngineConfiguration">
  ...
  <property name="beans"> #2-S
    <map> // entry的key属性就是变量名称
      <entry key="myBean" value-ref="myBean"></entry>
    </map>
  </property>
</bean> #2-E
<bean id="myBean" class="me.kafeitu.activiti.chapter7.expression.MyBean" /> #3
```

[1] <http://juel.sourceforge.net/index.html>, JUEL不仅作为一个开源的EL解析工具，而且是JSP2.1（JSR-245规范）中EL的解析器，已经被集成到了JAVAE5中。

7.4 监听器

在4.7节中已经大致了解了监听器的作用及基本的配置方式，代码清单7-7的#2-1处已经涉及了任务监听器的使用，本节将以实例方式讲解监听器的具体使用。

 **建议** 读者可以回顾一下4.7节的内容，本节的实例以4.7节的内容为基础。

对于监听器的配置已经陆续涉及了，本节的主要内容是如何在监听器中注入自定义字段，以及如何使用Spring代理监听器对象。

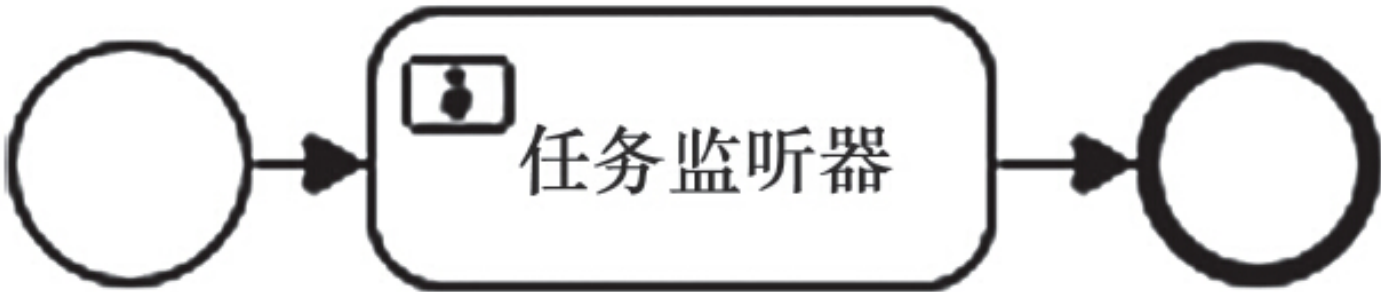


图7-4 监听器示例流程

为了更好地理解监听器与字段注入如何使用，下面结合图7-4的流程来讲解。

从图7-4中看不出这个流程要做什么，代码清单7-12列出了该流程的XML代码。

代码清单7-12 图7-3对应的XML

```
<process id="listener" name="listener">
  <extensionElements>
    <activiti:executionListener event="start" // 以class方式配置流程启动监听器 #1
      class="me.kafeitu.activiti.chapter7.listener.ProcessStartExecutionListener"/>
    <activiti:executionListener event="end" #2
      // 以delegateExpression方式配置流程结束监听器
      delegateExpression="${endListener}"></activiti:executionListener>
  </extensionElements>
  <startEvent id="startevent1" name="Start"></startEvent>
  <endEvent id="endevent1" name="End"></endEvent>
  <userTask id="usertask1" name="任务监听器" activiti:assignee="henryyan">
    <extensionElements>
      <activiti:taskListener event="create" // 为该任务添加任务创建监听器 #3
        class="me.kafeitu.activiti.chapter7.listener.CreateTaskListener">
      <activiti:field name="content"> // 为监听器注入字段content #4-S
        <activiti:expression>Hello, ${name}</activiti:expression>
      </activiti:field> #4-E
      <activiti:field name="task"> // 为监听器注入字段引擎内置变量task #5-S
        <activiti:expression>${task}</activiti:expression>
      </activiti:field> #5-E
      <activiti:taskListener>
        <activiti:taskListener event="assignment" // 配置任务分配监听器 #6
          delegateExpression="${assignmentDelegate}"/>
      </activiti:taskListener>
    </extensionElements>
  </userTask>
</process>
```

代码清单7-12中的#1、#2、#3处分别配置了三个监听器，其中#1、#2处把监听器绑定在流程实例的创建与结束事件上，#1使用class方式配置了一个类，而#2使用表达式（delegateExpression）方式配置，在运行时动态指定。

#3处把监听器绑定在用户任务“任务监听器”的create事件上，当任务被创建时执行监听器配置的class属性的类，并且在调用监听器时注入两个字内容content和task。

在开始单元测试之前先了解一下为监听器注入字段如何配置，图7-5展示了在Activiti Designer插件中如何设置监听器的字段注入。

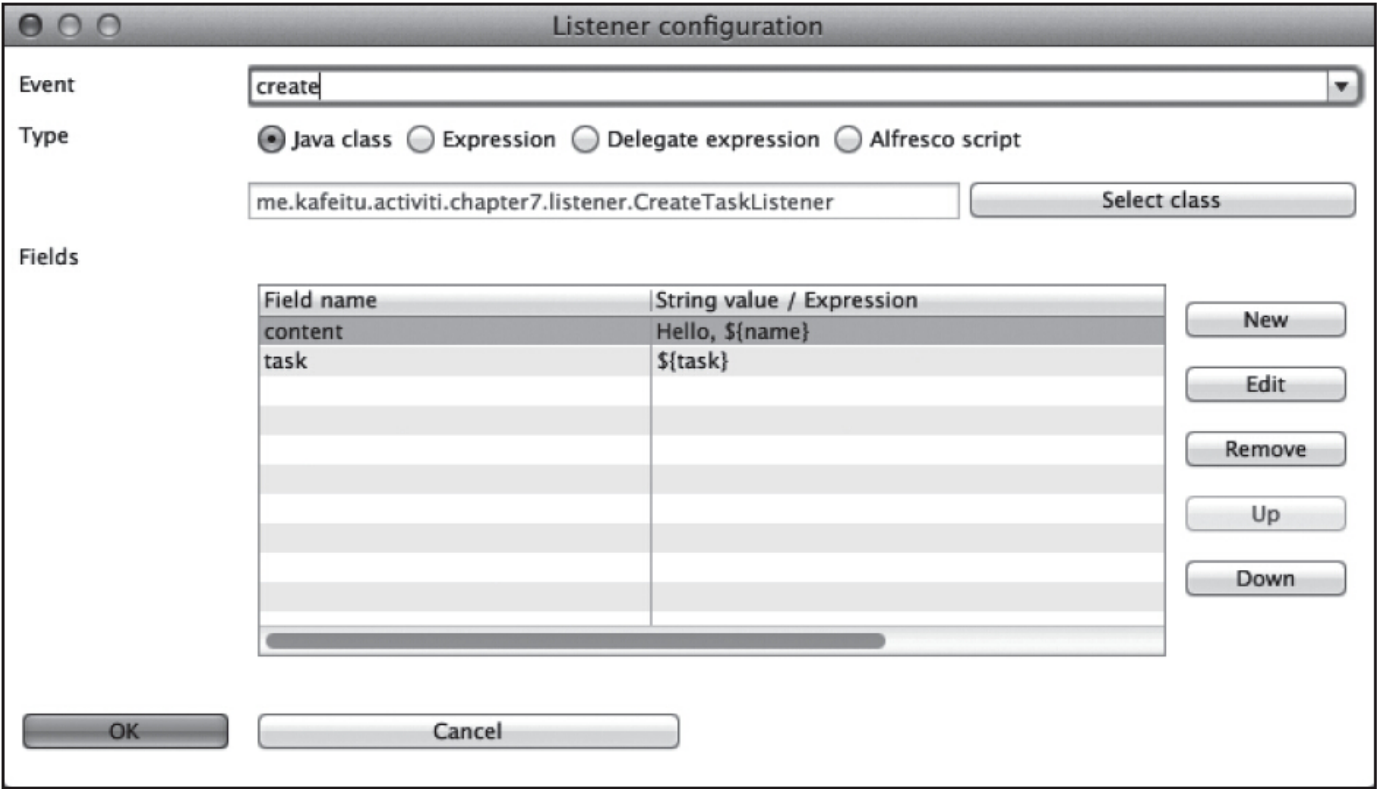


图7-5 使用Eclipse插件Activiti Designer为监听器设置注入字段

经过图7-5的配置后，XML代码如代码清单7-12的#3、#4、#5处所示，监听器实现类中必须有同名的且类型为Expression的字段，可以选择性地提供字段的setter方法，如果没有setter方法，引擎会通过反射机制设置变量（即设置访问权限为private）值。

字段的值可以是一个纯文本（content字段的“Hello, ”部分），也可以是表达式（content字段的\${name}），并且两者可以混用；如果字段仅仅设置了纯文本，那么在XML中将使用<activiti:string>标签代替<activiti:expression>，但是在监听器的类中还是需要使用Expression作为字段的类型。

在了解了监听器与字段注入的配置方式后，下面针对代码清单7-12的流程定义使用单元测试的方式来学习监听器何时被调用，字段如何注入。

代码清单7-13列出了流程启动与结束监听器的实现。

代码清单7-13 流程启动与结束监听器实现类

```
// 流程实例启动监听器实现类
public class ProcessStartExecutionListener implements ExecutionListener {           #1
    public void notify(DelegateExecution execution) throws Exception {
        execution.setVariable("setInStartListener", true);
        System.out.println(this.getClass().getSimpleName() + " , " + execution.getEventName());
    }
}
// 流程实例结束监听器实现类
public class ProcessEndExecutionListener implements ExecutionListener, Serializable {           #2 public void notify(Deleg
    execution.setVariable("setInEndListener", true);
    System.out.println(this.getClass().getSimpleName() + " , " + execution.getEventName());
}
}
```

流程实例级别的监听器均需要实现接口org.activiti.engine.delegate.ExecutionListener，此接口仅有一个方法且传入一个org.activiti.engine.delegate.DelegateExecution对象作为参数，参数execution和表7-2中的execution是一个对象，所以作用也是一样的。

任务（Task）级别的监听器与流程实例级别的监听器类似，只不过前者的参数类型有所不同而已。代码清单7-14列出了任务监听器，需要注意的是，虽然名称为“CreateTaskListener”但是不代表就是一个任务创建监听器，而是需要在流程定义文件中与任务事件进行绑定。

代码清单7-14 任务监听器实现类

```
public class CreateTaskListener implements TaskListener {
    private Expression content; // 表达式注入字段的类型必须为Expression
    private Expression task;
    public void notify(DelegateTask delegateTask) {
        System.out.println(task.getValue(delegateTask));
        delegateTask.setVariable("setInTaskCreate", delegateTask.getEventName() + " , "
            + content.getValue(delegateTask));
        System.out.println(delegateTask.getEventName() + " , 任务分配给: "
            + delegateTask.getAssignee());
        delegateTask.setAssignee("jenny"); // 重新分配任务给jenny
    }
    // 字段content、task的setter方法可以省略
}
// 监听任务分配
public class TaskAssigneeListener implements TaskListener, Serializable {
    public void notify(DelegateTask delegateTask) {
        System.out.println("任务分配给: " + delegateTask.getAssignee());
    }
}
}
```

代码清单7-14的CreateTaskListener实现了接口org.activiti.engine.delegate.TaskListener，和流程实例级别的监听器类似，都需要实现一个notify()方法，唯一不同的是参数类型不同。但是两者的notify()方法的参数对象均可以通过调用getEventName()方法获取触发监听器的事件类型，DelegateExecution的事件类型取值范围为start、end，DelegateTask的事件类型取值范围为create、assignment、complete。

代码清单7-15对配置的监听器执行结果进行了测试，以方便读者理解执行过程。

代码清单7-15 监听器单元测试

```
public class ListenerTest extends AbstractTest {
    @Test
    @Deployment(resources = { "diagrams/chapter7/listener/listener.bpmn" })
    public void testListener() {
        Map<String, Object> variables = new HashMap<String, Object>();
        // 创建一个监听器实例并设置到变量中
        variables.put("endListener", new ProcessEndExecutionListener());           #1-S
        variables.put("assignmentDelegate", new TaskAssigneeListener());           #1-E
        variables.put("name", "Henry Yan");
        ProcessInstance processInstance = runtimeService
            .startProcessInstanceByKey("listener", variables);
        // 校验是否执行了启动监听
        String processInstanceId = processInstance.getId();
        assertTrue((Boolean) runtimeService.getVariable(processInstanceId, "set-InStartListener"));
        // 校验任务监听器是否被执行
        Task task = taskService.createTaskQuery().processInstanceId(processInstanceId)
            .taskAssignee("jenny").singleResult(); // 监听器已经把任务重新分配给 jenny
        String setInTaskCreate = (String) taskService.getVariable(task.getId(),
            "setInTaskCreate"); // #2
        assertEquals("create, Hello, Henry Yan", setInTaskCreate); // #2
        taskService.complete(task.getId());
        // 流程结束后查询变量
        List<HistoricVariableInstance> list = historyService.createHistoricVariableInstanceQuery()
            .processInstanceId(processInstanceId).list(); // 查询历史变量
        boolean hasVariableOfEndListener = false;
        for (HistoricVariableInstance variableInstance : list) {
            if (variableInstance.getVariableName().equals("setInEndListener")) {
                hasVariableOfEndListener = true;
            }
        }
        assertTrue(hasVariableOfEndListener); // 校验流程结束监听器是否被执行
    }
}
```

代码清单7-15针对每个监听器被执行的情况进行了验证，#1处需要注意流程结束监听器是动态指定的一个对象，而不是像流程启动监听器一样显示地执行一个class，当然可以利用7.3.3的方式在引擎配置中注入流程结束监听器Bean。

单元测试执行完成后可以在控制台看到类似如图7-6所示的输出结果，可以看出任务监听器TaskAssigneeListener被执行了2次。

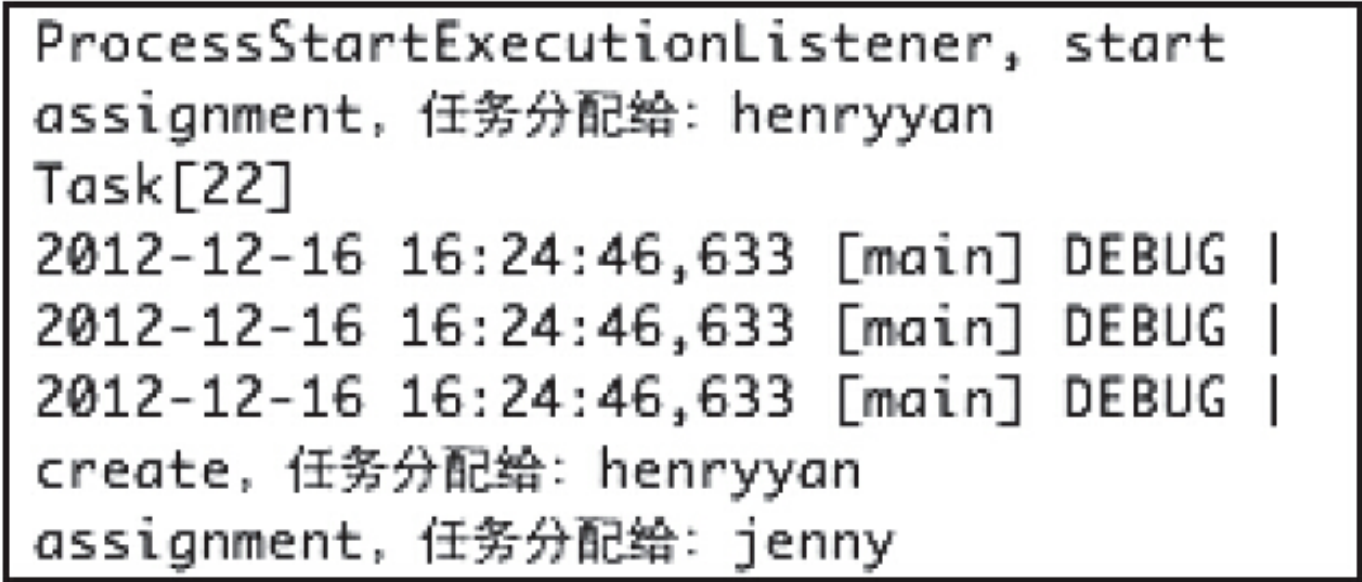


图7-6 ListenerTest执行结果

从7.3节和7.4节对表达式以及监听器的学习可以看出，在Activiti中可以通过各种方法灵活运用实现业务逻辑的处理，这些都得益于JUEL的支持，所以在实际应用中要选择适合团队或项目的方式，利用表达式和监听器实现业务逻辑的处理。

7.5 Spring容器集成应用实例

上一章介绍了任务表单，包括动态表单和外置表单，其实在实际应用中还有一种普通表单（暂且这么定义吧），直接把表单的内容写在表现层（JSP、JSF、HTML等）文件中，一个用户任务对应一个页面（或者将内容放在一个文件中，根据节点判断显示哪部分内容）。这样的做法适用于业务相对比较固定、业务比较复杂、流程相对固定但表现层变化比较多的情况。

一般使用普通表单时业务数据和流程数据是分离的，这就牵扯到一个事务问题。假设这样一个场景，用户完成一个任务并更新业务数据（更改了表单），再假设此时任务已经执行了完成（complete）操作，但是在更新表单数据时失败了，所以更改执行的任务完成操作就要回滚，此时就要保证数据的一致性。

要解决这样的问题就要统一事务管理，保证Activiti与业务数据操作在同一个事务中执行。7.1.2节介绍的Spring正是解决此问题的途径，由Spring提供统一的事务管理器并注入到Activiti引擎和业务层（Service层）中。

本节将综合本章内容对事务问题的处理采用普通表单方式设计一个和第6章功能相同的请假流程，主要针对在数据和表单都分离的情况下如何使两者结合。

7.5.1 业务建模

首先对数据进行建模，请假业务表的表结构如图7-7所示，从表结构上看和第6章中的表单字段基本类似。

Run (Ctrl+Enter)

Clear

SQL statement:

show COLUMNS from leave

show COLUMNS from leave;

FIELD	TYPE	NULL	KEY	DEFAULT
ID	BIGINT(19)	NO	PRI	(NEXT VALU
PROCESS_INSTANCE_ID	VARCHAR(64)	YES		NULL
USER_ID	VARCHAR(20)	NO		NULL
START_TIME	TIMESTAMP(23)	NO		NULL
END_TIME	TIMESTAMP(23)	NO		NULL
LEAVE_TYPE	VARCHAR(20)	YES		NULL
REASON	VARCHAR(2000)	YES		NULL
APPLY_TIME	TIMESTAMP(23)	NO		NULL
REALITY_START_TIME	TIMESTAMP(23)	YES		NULL
REALITY_END_TIME	TIMESTAMP(23)	YES		NULL

图7-7 请假业务表结构

对业务实体的映射使用JPA标准的注解方式，即使用目前企业应用中比较流行的ORM框架Hibernate作为持久层的实现，就是7.1.2节中介绍的Spring的Activiti引擎并统一管理业务与流程的事务。

本节的内容将全部使用Spring代理DAO、Manager以及Spring MVC的Controller，其中DAO和Manager用来针对请假实体的CRUD操作。另外还要创建一个Leave-WorkflowService类来处理流程相关操作，例如流程启动、读取任务列表等。

DAO及Manager的类图如图7-8所示，LeaveWorkflowService的代码见代码清单7-16所示。

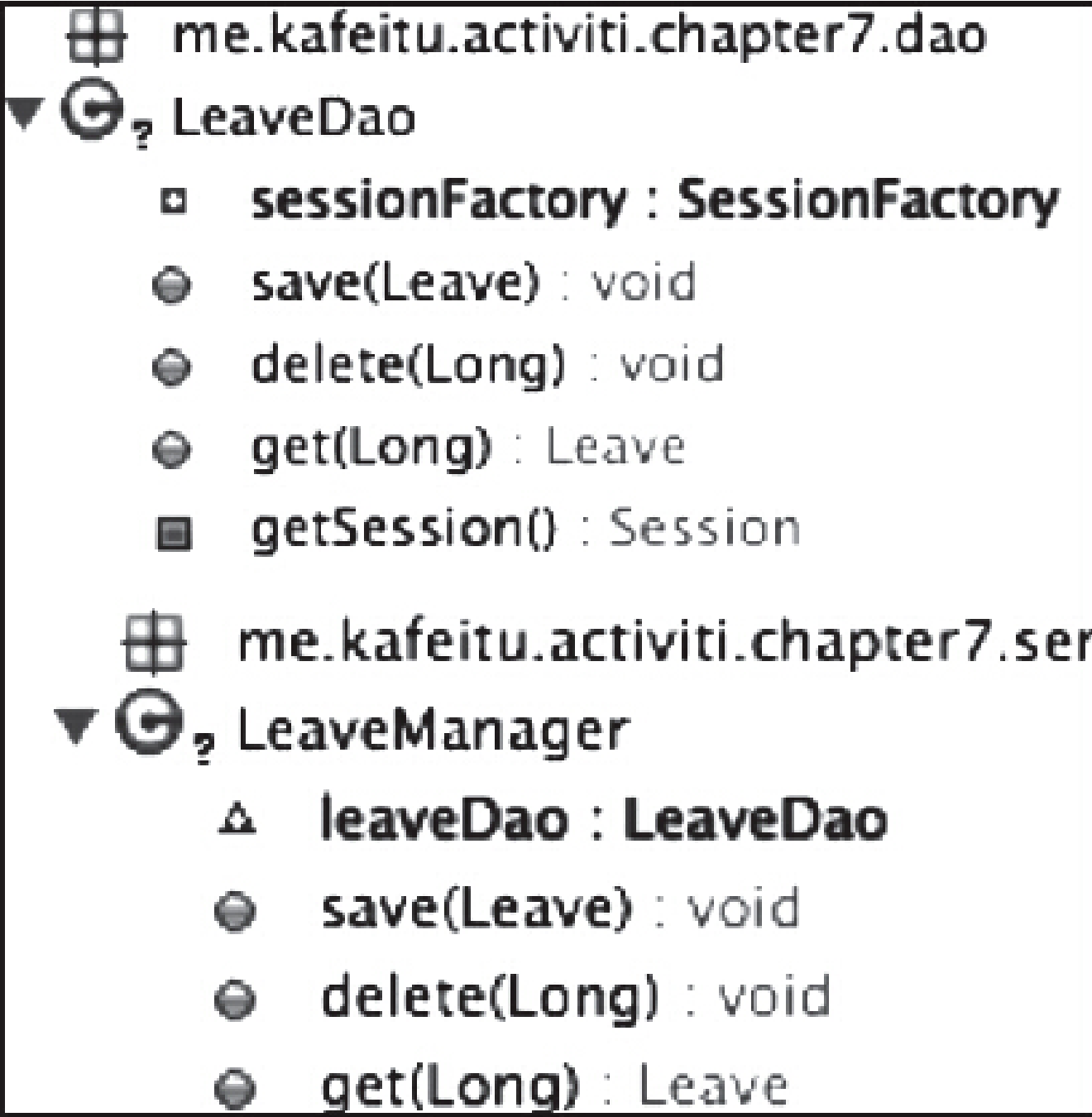


图7-8 请假实体相关DAO与Manager类图

7.5.2 启动流程

图7-9中的流程定义列表包含了“请假流程-普通表单”流程定义，通过在Spring实现的流程引擎中设置“deploymentResources”属性自动部署。

部署流程资源							
Choose File No file chosen Submit							
流程定义ID	部署ID	流程定义名称	流程定义KEY	版本号	XML资源名称	图片资源名称	操作
leave:1:1704	1701	请假流程-普通表单	leave	1	leave.bpmn	leave.png	删除 启动

图7-9 包含了“请假流程-普通表单”的流程定义列表

 提示 本节涉及的流程定义文件位于“chapter7/leave.bpmn”。


```

@Service
@Transactional // 启用事务管理
public class LeaveWorkflowService {
    private Logger logger = LoggerFactory.getLogger(getClass());
    @Autowired
    LeaveManager leaveManager; // 注入业务实体管理对象
    @Autowired
    private IdentityService identityService; // 注入Activiti相关Service
    @Autowired
    private RuntimeService runtimeService;
    // 保存请假实体并启动流程
    public ProcessInstance startWorkflow(Leave entity,
        String userId, Map<String, Object> variables) { //
        if (entity.getId() == null) {
            entity.setApplyTime(new Date());
            entity.setUserId(userId);
        }
        leaveManager.save(entity); // 持久化请假实体 #1
        String businessKey = entity.getId().toString(); // 实体保存后的ID, 作为流程的业务Key
        // 用来设置启动流程的人员ID, 引擎会自动把用户ID保存到activiti:initiator中
        identityService.setAuthenticatedUserId(userId);
        ProcessInstance processInstance = runtimeService // 启动流程时设置业务Key
            .startProcessInstanceByKey("leave", businessKey, variables); #2
        String processInstanceId = processInstance.getId();
        entity.setProcessInstanceId(processInstanceId); // 建立双向关系 #3
        logger.debug("start process of {key={}, bkey={}, pid={}, variables={}}",
            new Object[] { "leave", businessKey, processInstanceId, variables });
        return processInstance;
    }
}

```

代码清单7-16有两处和以往不同，业务表独立但又需要和流程建立关联关系，所以首先把业务实体持久化（#1处），然后利用持久化后得到的业务对象ID作为启动流程的参数（#2处的businessKey），如此流程中保存了业务的主键ID。但是为了方便以后从业务层面查询流程数据，在#3处的业务实体中设置了流程实例ID，如此就建立了业务与流程的双向关联关系。

当然，在代码清单7-16的#3处执行的建立双向关系是可选的，这样做的目的是在查询时可以明确知道该业务对应的流程数据ID，假如没有在企业业务中保存流程实例的ID，那么两者联合查询要以流程为入口，如此在性能方面就大打折扣了。这样做还有另外一个目的，那就是可以区分业务数据的状态，例如在有些系统中会区分已启动流程和未启动流程的数据，根据业务数据的“PROCESS_INSTANCE_ID”字段是否为空可以很容易地查询不同状态的数据。

从代码清单7-16可以看出，因为LeaveWorkflowService是受事务控制的，所以本节开始的那些假设都可以正确地处理，如果实体保存了但是启动流程的时候失败，则全部回滚。

在本节开始时就提到了使用普通表单时需要包裹内容保存在一个单独的文件中，图7-10是针对请假流程的启动设计的一个自定义表单。对应的JSP代码可参见chapter7-spring/src/main/webapp/WEB-INF/views/chapter7/leave/leaveApply.jsp。

The screenshot shows the 'Activiti Explorer' interface with a navigation bar containing '首页' (Home), '请假（普通表单）' (Leave - Standard Form), '任务列表' (Task List), and '管理' (Management). The main content area is titled '请假申请' (Leave Application). The form contains the following fields:

- 请假类型:** A dropdown menu with '事假' (Sick Leave) selected.
- 开始时间:** Two input fields showing '2012-12-21' and '09:10'.
- 结束时间:** Two input fields showing '2012-12-28' and '09:10'.
- 请假原因:** A text area containing '家中有事' (Family matters).
- 启动流程:** A button with a play icon and the text '启动流程' (Start Process).

图7-10 请假（普通表单）的申请页面

在点击图7-10的“启动流程”按钮后，将提交表单至“chapter7/leave/start”，对应的Controller见代码清单7-17。

代码清单7-17 请假流程的Controller

```

@Controller
@RequestMapping(value = "/chapter7/leave")
public class LeaveController {
    @Autowired
    private LeaveWorkflowService leaveService;
    @RequestMapping(value = "start", method = RequestMethod.POST)
    public String startWorkflow(Leave leave, HttpSession session, #1
        RedirectAttributes redirectAttributes) {
        User user = UserUtil.getUserFromSession(session);
    }
}

```



```

        Map<String, Object> variables = new HashMap<String, Object>();          #2
        ProcessInstance processInstance = leaveService
            .startWorkflow(leave, user.getId(), variables);          #3
        redirectAttributes.addFlashAttribute("message", "流程已启动, 流程ID: "
            + processInstance.getId());
    }

```

在代码清单7-17的#1处，第一个参数是“leave”（Spring MVC会根据表单的内容自动封装对象）；在#2处创建一个Map对象来设置启动流程的变量集合，此处没有变量，所以没有设置值。

7.5.3 任务读取

和第6章的例子一样，在流程启动之后需要办理任务，所以需要有一个任务列表读取当前人的待办任务。当然在第6章我们已经设计了一个任务列表，可以胜任读取待办任务的功能，但是目前的需求不能满足普通表单的要求了，图7-11才是最终想要的结果。

这就要求要把请假实体和任务对象相关联，也只有这样才能输出图7-11这样的列表，可以很清楚地了解一条任务的业务信息。

申请人	类型	请假时间	请假原因	任务ID	任务名称	流程实例ID	流程定义ID	任务创建时间	操作
kermit	公休	2012-12-21 09:10:00.0至 2012-12-28 09:00:00.0	家中有事	105	部门领导审批	101	leave:1:4	Thu Dec 20 22:06:25 CST 2012	<button>办理</button>

图7-11 待办任务列表综合业务和任务信息

在第6章任务列表查询的基础上稍加改造即可得到图7-11的任务列表。在代码清单6-10中已经实现了任务的读取，我们只要在查询完所有任务之后再根据启动流程时传入的业务ID查询请假记录并关联两者即可。

代码清单7-18在代码清单6-10的基础上添加了一段代码，用来根据流程数据的业务ID查询关联的业务对象。

代码清单7-18 关联读取业务与任务数据

```

public class LeaveWorkflowService {
    @Transactional(readOnly = true)
    public List<Leave> findToDoTasks(String userId) {
        ... // 省略了查询任务的代码，参加代码清单6-10
        // 根据流程的业务ID查询实体并关联
        for (Task task : tasks) {
            String processInstanceId = task.getProcessInstanceId();
            ProcessInstance processInstance = runtimeService.createProcessInstanceQuery()
                .processInstanceId(processInstanceId).singleResult();
            String businessKey = processInstance.getBusinessKey(); // 获取启动流程时的业务Key
            Leave leave = leaveManager.get(new Long(businessKey));
            leave.setTask(task); // 在Leave类中添加了task属性的getter和setter
            results.add(leave);
        }
        return results;
    }
}

```

对应的LeaveController类也添加了一个读取任务列表的方法，见代码清单7-19。

代码清单7-19 向请假流程的Controller添加读取任务列表的方法

```

@Controller
@RequestMapping(value = "/chapter7/leave")
public class LeaveController {
    @Autowired
    private LeaveWorkflowService leaveService;
    @RequestMapping(value = "task/list")
    public ModelAndView taskList(HttpSession session) {
        ModelAndView mav = new ModelAndView("/chapter7/leave/leave-task-list");
        String userId = UserUtil.getUserFromSession(session).getId();
        List<Leave> results = leaveService.findToDoTasks(userId);
        mav.addObject("records", results);
        return mav;
    }
}

```

7.5.4 任务办理

任务办理页面和请假申请一样需要针对每个任务设置不同的表单，和外置表单一样，每一个任务都设置了不同的formkey属性，在任务办理时可以动态获取。

使用普通表单也是类似的原理，首先针对每个任务设计好对应的表单及业务逻辑处理，然后单击图7-11中的“办理”按钮页面跳转到当前要办理的任务的对应页面（根据流程定义文件中的用户任务ID属性），如此就可以和外置表单一样动态选择表单。

图7-12的目录结构清楚地展示了根据任务ID属性命名的JSP文件。



在图7-12中，加边框部分以“task-”开头的文件就是针对不同的用户任务定制的显示内容，当办理任务时，请求Spring MVC的Controller（或者Struts2的Action等）动态返回视图的名称（以“task-”开头或者自定义的一个规则）。

代码清单7-20在LeaveController中添加了一个方法“showTaskView”，实现了动态返回视图的功能，并在视图对象中包含了请假实体及任务对象。

代码清单7-20 请假流程——动态显示任务表单

```
@RequestMapping(value = "task/view/{taskId}")
public ModelAndView showTaskView(@PathVariable("taskId") String taskId) {
    Task task = taskService.createTaskQuery().taskId(taskId).singleResult();
    // 读取流程实例对象并获取启动流程时传入的businessKey属性
    ProcessInstance processInstance = runtimeService.createProcessInstanceQuery()
        .processInstanceId(task.getProcessInstanceId()).singleResult();
    Leave leave = leaveManager.get(new Long(processInstance.getBusinessKey()));
    ModelAndView mav = new ModelAndView( // 动态拼接视图名称
        "/chapter7/leave/task-" + task.getTaskDefinitionKey());
    mav.addObject("leave", leave); // 设置leave对象，可以在视图中获取属性
    mav.addObject("task", task); // 设置task对象，在视图中可以读取任务相关属性
    return mav;
}
```

单击如图7-10中的“办理”按钮后页面跳转请求了代码清单7-20的处理方法，最后页面跳转至“task-deptLeaderVerify.jsp”，然后显示如图7-13所示的办理界面。

部门领导审批

申请人: kermit
申请时间: 2012-12-23 01:52:45.989
请假类型: 公休
开始时间: 2012-12-21 09:00:00.0
结束时间: 2012-12-28 17:50:00.0
请假原因: 家中有事

审批意见: 同意

完成任务

图7-13 请假流程的“部门领导审批”节点办理界面

在流程定义文件中，完成“部门领导审批”用户任务需要经过一个排他分支（4.4.1节的，领导同意申请时的条件为“\${deptLeaderApproved}”，不同意时的条件为“\${! deptLeaderApproved}”，由7.3节获取的对表达式的理解可以知道，当完成此任务时需要传入一个名称为“deptLeaderApproved”、类型为Boolean（布尔型）的变量。第6章的条件判断表达式根据一个字符串进行比较“\${deptLeaderApproved}==‘true’”，所以deptLeaderApproved的类型应该为String，但是本节要求使用Boolean，因此在任务完成时需要进行类型转换。如果完成任务时涉及更新业务实体的属性，还要保证字段与实体的属性不冲突（有可能表单的字段属性名称相同）。代码清单7-21针对普通表单的任务完成做了简单的实现。

代码清单7-21 请假流程——普通表单任务完成方法实现

```
@RequestMapping(value = "task/complete/{id}")
public String complete(@PathVariable("id") String taskId, @RequestParam(value = "saveEntity", required = false) String saveEntity,
    @ModelAttribute("preloadLeave") Leave leave, RedirectAttributes redirectAttributes) {
    boolean saveEntityBoolean = BooleanUtils.toBoolean(saveEntity); // 是否更新业务实体
    Map<String, Object> variables = new HashMap<String, Object>(); #1-S
    Enumeration<String> parameterNames = request.getParameterNames();
    try {
        while (parameterNames.hasMoreElements()) {
            String parameterName = (String) parameterNames.nextElement();
            if (parameterName.startsWith("p_")) {
                // 参数结构: p B name, p为参数的前缀, B为类型, name为属性名称
                String[] parameter = parameterName.split("_");
                if (parameter.length == 3) {
```

```
String paramValue = request.getParameter(parameterName); // 默认为字符串型
Object value = paramValue;
if (parameter[1].equals("B")) { // 针对Boolean类型进行处理
    value = BooleanUtils.toBoolean(paramValue);
} else if (parameter[1].equals("DT")) { // 针对日期类型进行处理
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm");
    value = sdf.parse(paramValue);
}
variables.put(parameter[2], value);
}
}
} #1-E
leaveService.complete(leave, saveEntityBoolean, taskId, variables); #2
redirectAttributes.addFlashAttribute("message", "任务已完成");
} catch (Exception e) {
    logger.error("error on complete task {}", variables={}, new Object[] { taskId, variables, e });
    request.setAttribute("error", "完成任务失败");
}
return "redirect:/chapter7/leave/task/list";
}
@Service
@Transactional
public class LeaveWorkflowService {
    // 任务完成时通过saveEntity判断是否保存leave对象
    public void complete(Leave leave, Boolean saveEntity, String taskId, Map<String, Object> variables)
    {
        if (saveEntity) {
            leaveManager.save(leave);
        }
        taskService.complete(taskId, variables);
    }
}
```

代码清单7-21先在#1处根据一个自定义的规则判断来自前端的请求参数，如果和预期规则相匹配，则将其加入到任务完成时的变量集合中，对应地，在JSP中设置了根据变量规则定义的属性名称。

```
<label for="deptLeaderApproved">审批意见: </label>
<div class="controls">
    <select name="p_B_deptLeaderApproved" id="deptLeaderApproved">
        <option value="true">同意</option>
        <option value="false">拒绝</option>
    </select>
</div>
```

代码清单7-21在#2处调用LeaveWorkflowService类的complete方法，并且根据变量saveEntity的状态判定是否需要更新实体来避免多余的数据库Update操作。

对于用户任务“部门领导审批”和“人事审批”不需要更新业务实体，但是对于“调整申请”任务就需要在任务完成的同时更新业务实体，办理界面如图7-14所示。

调整申请

请假类型:

公休

开始时间:

2012-12-21

02:40

结束时间:

2012-12-28

02:40

请假原因:

家中有事

是否继续申请:

重新申请

✔ 完成任务

图7-14 请假流程——普通表单的“调整申请”节点办理界面

从代码清单7-21中可以看出，如果变量“saveEntity”为“true”，那么可以把表单的内容在任务完成之前更新，并且因为“LeaveWorkflowService”是受事务控制的（通过注解@Transactional标记），所以保存实体和任务完成具有事务的一致性。我们可以在展示层的“task-modifyApply.jsp”的表单中添加一个隐藏域来设置saveEntity的值。

```
<input type="hidden" name="saveEntity" value="true" />
```

通过以上代码完成任务“部门经理审批”以及“人事审批”之后，任务回归到申请人，办理界面如图7-15所示。

销假

申请人:

kermit

申请时间:

2012-12-23 01:52:45.989

请假类型:

公休

开始时间:

2012-12-21 02:40:00.0

结束时间:

2012-12-28 02:40:00.0

请假原因:

家中有事

实际开始时间:

2012-12-21

02:40

实际结束时间:

2012-12-28

02:40

✔ 完成任务

图7-15 请假流程——普通表单“销假”办理界面

图7-15中的“实际开始时间”和“实际结束时间”字段属于Leave实体，所以在任务完成时需要更新实体，不过在用户任务“调整申请”中已经演示过了，所以我们利用另外一种方法来实现任务监听器。

在本例的流程定义文件的“销假”任务中添加一个“complete”事件类型的任务监听器，当任务完成时把页面传递的变量更新到Leave实体。

```
<userTask id="reportBack" name="销假" activiti:assignee="${applyUserId}">
  <extensionElements>
    <activiti:taskListener event="complete"
      delegateExpression="${reportBackEndProcessor}"/>
  </extensionElements>
</userTask>
```

代码清单7-22列出了“销假”任务的完成时监听器，可以根据流程变量更新Leave实体并持久化，当然为了统一事务管理需要使用Spring代理。

代码清单7-22 请假流程——普通表单中销假任务完成监听器

```
@Component
@Transactional
public class ReportBackEndProcessor implements TaskListener {
    @Autowired
    LeaveManager leaveManager;
    public void notify(DelegateTask delegateTask) {
        String businessKey = delegateTask.getExecution().getProcessBusinessKey();
        Leave leave = leaveManager.get(new Long(businessKey));
        Object realityStartTime = delegateTask.getVariable("realityStartTime");
        leave.setRealityStartTime((Date) realityStartTime);
        Object realityEndTime = delegateTask.getVariable("realityEndTime");
        leave.setRealityEndTime((Date) realityEndTime);
        leaveManager.save(leave);
    }
}
```

至此一个基于Spring使用普通表单，并且业务和流程数据库分离的应用告一段落，其中涉及了表达式、监听器、事务统一管理。

7.6 使用Spring注解初始化引擎

以上内容都是以XML方式配置引擎，不过目前一部分开发人员偏向于全注解的编程习惯，以及推崇“约定优于配置”的思想，所以Spring 3.0以及Spring 4.0版本都强化了注解的作用，本节将介绍如何

使用注解的方式初始化引擎。

7.6.1 使用@EnableActiviti注解

在Spring 3.0中加入了很多有用的注解，使用全注解可以初始化Spring上下文对象（ApplicationContext）或者注入Bean。Spring团队的Josh Long从Activiti创立开始就一直致力于如何让Activiti与Spring更完美的结合，前面章节介绍了如何通过XML方式配置引擎参数并初始化引擎，本节将介绍如何使用Spring注解初始化流程引擎并注入Service接口。

Spring的@[Configuration](#)注解可以定义在一个类上，表示这个类被用做配置对象。有关该注解的详细使用方法可参看Spring的用户手册或者API文档，其中均有详细介绍。activiti-spring模块默认提供了基于注解的实现类[ActivitiConfiguration](#)，可以在一个类上标记@[Configuration](#)和@[EnableActiviti](#)注解来启用该类初始化引擎，可以创建一个AnnotationConfigApplicationContext对象获取引擎的Service接口，具体的代码见代码清单7-23所示。

代码清单7-23 使用Spring注解初始化引擎并注入Bean

```
public class InitProcessEngineBySpringAnnotation {
    @Configuration // Spring的注解
    @EnableActiviti // activiti-spring模块的注解 #1
    public static class SpringAnnotationConfiguration {
        @Bean // 自定义数据源
        public DataSource dataSource() throws ClassNotFoundException { #2
            SimpleDriverDataSource ds = new SimpleDriverDataSource();
            ds.setDriverClass((Class<? extends Driver>) Class.forName("org.h2.Driver"));
            ds.setUrl("jdbc:h2:mem:iaia-chapter7;DB_CLOSE_DELAY=1000");
            ds.setUsername("sa");
            ds.setPassword("");
            return ds;
        }
    }
    public static void main(String[] args) {
        AnnotationConfigApplicationContext ctx =
            new AnnotationConfigApplicationContext(); #3-1
        ctx.register(SpringAnnotationConfiguration.class);
        ctx.refresh(); #3-2
        // 验证引擎是否成功初始化
        assertNotNull(ctx.getBean(ProcessEngine.class)); #4-1
        assertNotNull(ctx.getBean(RuntimeService.class));
        assertNotNull(ctx.getBean(TaskService.class));
        assertNotNull(ctx.getBean(HistoryService.class));
        assertNotNull(ctx.getBean(RepositoryService.class));
        assertNotNull(ctx.getBean(ManagementService.class));
        assertNotNull(ctx.getBean(FormService.class)); #4-2
    }
}
```

代码清单7-23中#1处定义了一个引擎配置类SpringAnnotationConfiguration，并使用@[Configuration](#)和@[EnableActiviti](#)注解标记。#2处的dataSource()方法返回一个数据源对象，该方法使用了@[Bean](#)注解标记，意思是指定引擎配置对象的数据源需要从该方法获取。需要特别注意的是，如果@[Bean](#)注解没有指定name属性，那么方法名必须与配置对象的属性名称保持一致，当然也可以通过@[Bean](#)(name="dataSource")来声明，该方法会返回一个名称为“dataSource”的Bean对象（此时方法名可以不与引擎配置对象的属性一致）。

在#3处创建一个AnnotationConfigApplicationContext对象并注册#1处标记的配置对象，通过刷新上下文对象来扫描SpringAnnotationConfiguration类中定义的Bean对象。最后在#4处从上下文对象中获取引擎的Service接口。

7.6.2 使用Spring Boot初始化引擎

Maven工具遵循“约定优于配置”的思想规定了项目目录的结构，2014年Spring团队发布的Spring Boot项目也正是借鉴了“约定优于配置”的思想。基于Spring Boot可以快速创建Spring应用或者服务。

和上一节介绍的注解方式类似，在初始化Spring时均有默认值，当然也可以覆盖配置参数。基于Spring Boot的初始化不用关心任何繁琐的XML配置，全部使用注解标记，并且可以不依赖外部容器的情况下创建独立的Java应用或者Web应用。

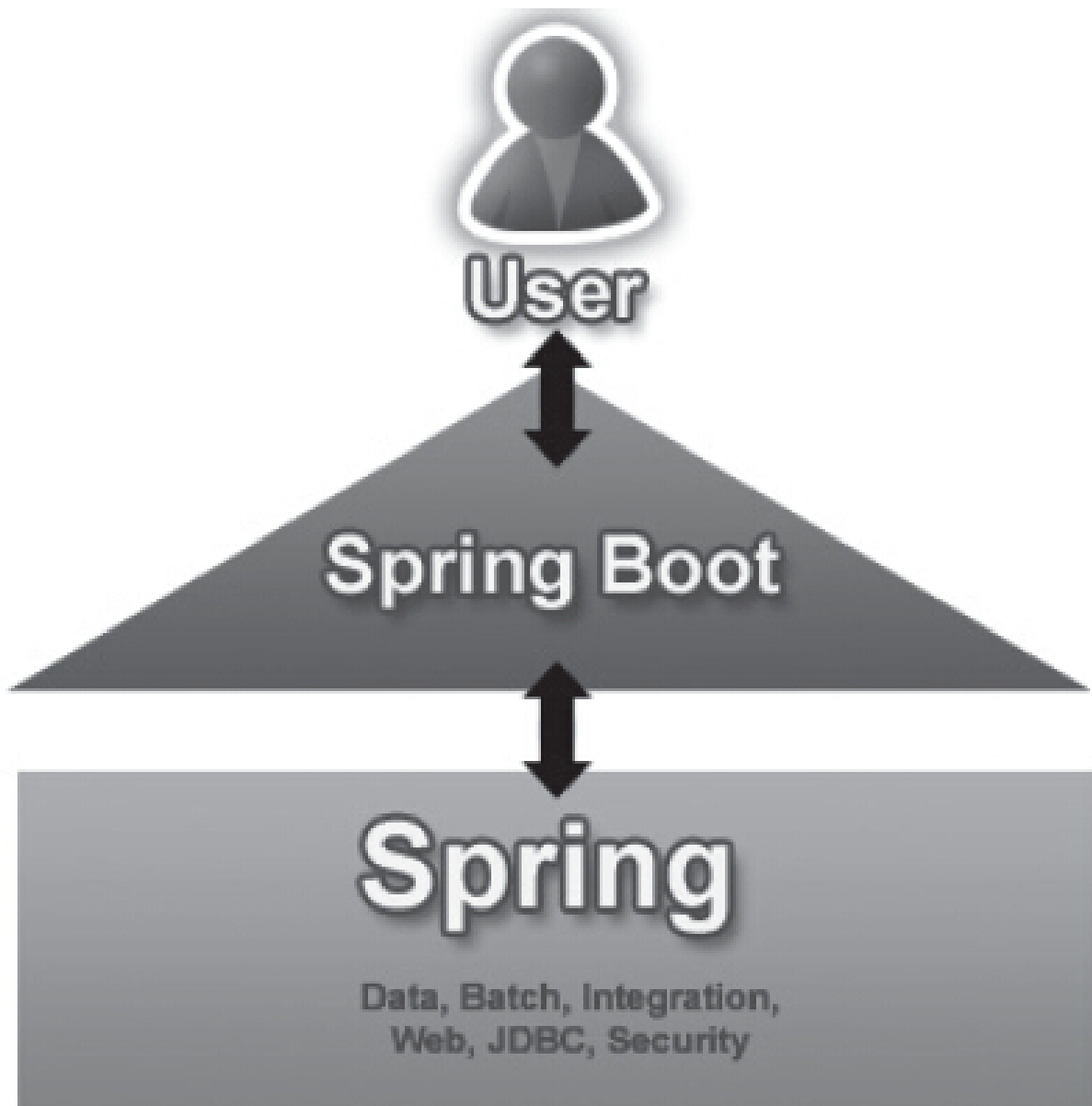


图7-16 Spring Boot在Spring生态中的位置

图7-16明显地说明了Spring Boot的作用及其在Spring生态中的位置。

在Activiti 5.16版本中也添加了对Spring Boot的支持，可以更方便地通过Spring初始化引擎以及获取Service接口对象。代码清单7-24列出了使用Spring Boot注解定义的引擎配置，代码清单7-24的类在chapter7-spring-boot项目中可以找到。

代码清单7-24 使用Spring Boot注解初始化引擎并注入Bean

```
public class ActivitiWithSpringBootApplication {
    @Configuration
    @ComponentScan
    @EnableAutoConfiguration // Spring Boot注解 #1
    public static class SpringBootConfiguration {
        // 这里可以用方法设置引擎的配置属性
    }

    public static void main(String[] args) throws Exception {
        // 启动应用 #2
        ApplicationContext ctx = SpringApplication.run(SpringBootConfiguration.class, args);
        assertNotNull(ctx.getBean(RuntimeService.class));
    }
}
```

代码清单7-24的#1处使用@EnableAutoConfiguration注解声明启用自动配置功能，由于Activiti的spring-boot模块中已经内置了支持Spring Boot的配置类并标注了相应的注解（配置类名称为ProcessEngineAutoConfiguration），所以当通过#2处的代码启动应用时Spring Boot会扫描并执行引擎配置对象的初始化动作。

图7-17列出了chapter7-spring-boot模块的目录结构，从图中可以看出在resources/processes目录中有一个名称为“simpleProcess.bpmn20.xml”的文件。activiti-spring-boot模块在初始化时会扫描该目录中扩展名为“*.bpmn20.xml”的文件并部署到引擎。

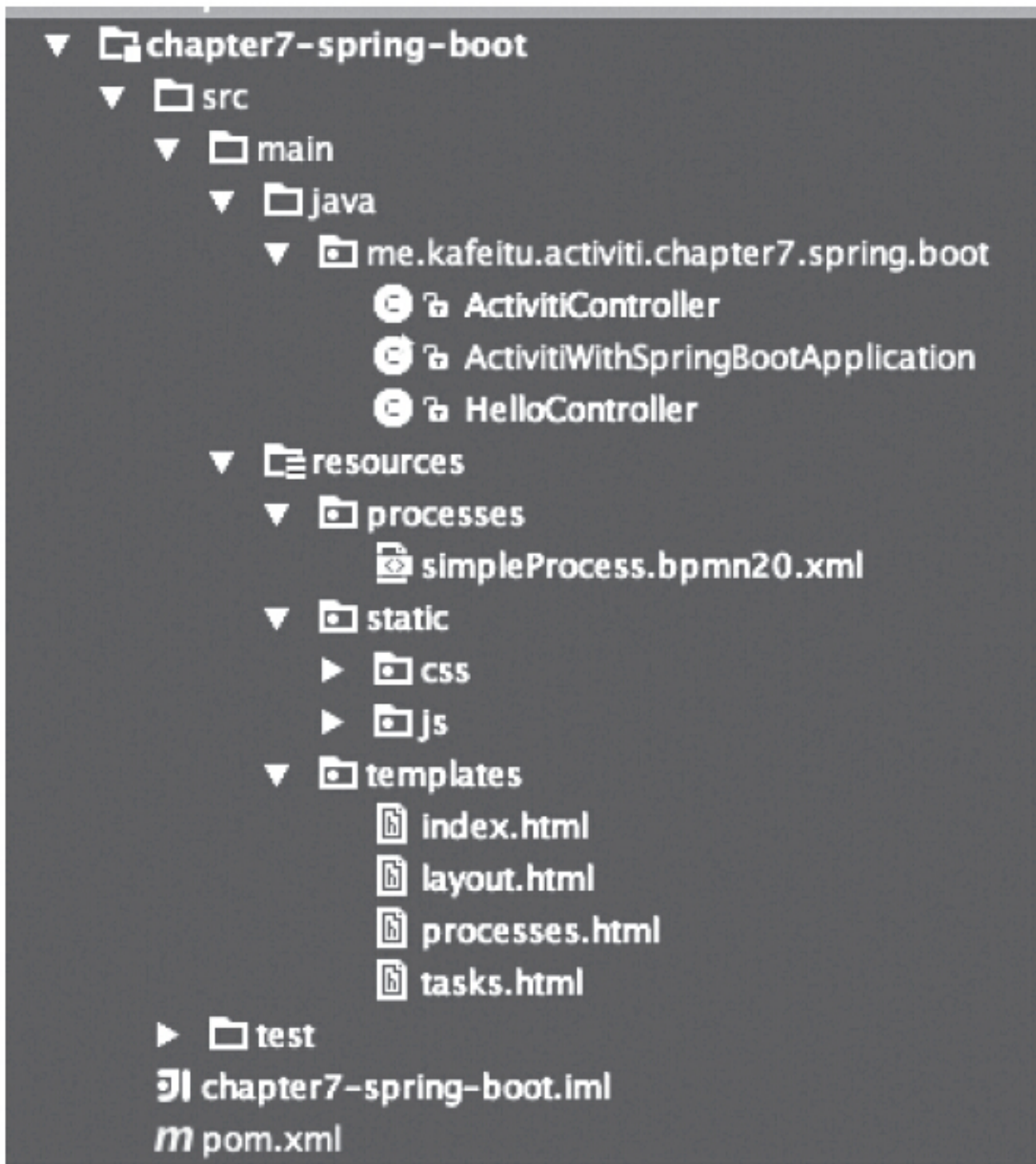


图7-17 chapter7-spring-boot模块的目录结构

图7-17中的ActivitiController类是Spring MVC的一个控制器，提供了读取流程定义、启动流程、查询任务、完成任务的简单功能，具体的实现代码可自行查看源码。

运行代码清单7-24中的Java类后通过Spring Boot的内嵌Web容器（Tomcat）启动了一个Web服务，相关的Maven依赖可参看chapter7-spring-boot模块中pom.xml文件中的定义，访问<http://localhost:8080>即可看到如图7-18所示的界面。

流程定义列表

localhost:8080/activiti/processes

Supervised

《Activiti实战》

引擎信息

流程定义

任务列表

流程定义列表

ID	KEY	NAME	OPTS
simpleProcess:1:4	simpleProcess	简单任务	启动

图7-18 自动部署的流程定义

通过Spring Boot注解初始化引擎后在Controller中可以通过注解方式注入引擎的Service接口，当然也可以在配置对象（代码清单7-24中#1处的SpringBootConfiguration类）中覆盖默认配置参数，相比XML方式更加的灵活。

7.7 CDI模块

除了使用Spring提供的注解实现依赖注入之外，在JAVA EE6版本中添加的上下文和依赖注入（Contexts and Dependency Injection for Java EE，简称CDI）标准允许开发人员在JAVA EE平台中使用CDI的注解实现上下文描述和依赖注入等扩展功能。CDI在JSR-299中有详细的描述，其中依赖注入使用了JSR-330（Dependency Injection for Java，简称DI）规范，该规范也被Spring、Guice等第三方框架所兼容。

CDI的实现方式有多种，例如JBoss Weld、Caucho CanDI、Apache OpenWebBeans，使用JAVA EE的CDI的注解描述Bean就可以在不同的实现平台之间实现移植。

 **提示** Activiti CDI模块仅在JBoss Weld环境中进行过测试，不同的实现平台之间可能有差异。

7.7.1 启动示例

本章使用的容器版本为JBoss AS 7.1.1，在本章示例（chapter7-cdi）的pom.xml文件中定义了“jboss.home”属性。在运行本章示例之前先更改JBoss的目录。运行本章示例的顺序如下。

- **启动JBoss容器**：执行JBoss容器中的bin/standalone.bat（Windows平台）或者bin/standalone.sh（Linux或者UNIX平台）。
- **部署示例到JBoss**：在示例代码根目录执行`mvn jboss-as:deploy`命令，使用了JBoss的Maven插件jboss-as-maven-plugin（先修改pom.xml文件中的jboss.home属性值为本地JBoss的根目录路径）。
- **访问示例**：在浏览器打开`http://localhost:8080/chapter7-cdi`。

本节示例代码中使用的视图技术是JSF，同时为了演示在页面中添加了三个菜单链接，如图7-19所示。接下来会——讲解如何使用Activiti CDI模块提供的功能操作流程数据。

localhost:8080/chapter7-cdi/index.xhtml

《Activiti实战》 第7章-CDI示例

菜单列表 - 流程定义 - 待办任务 - 变量列表	引擎属性 {next.dbid=2501, schema.history=create(5.16), schema.version=5.16}
--	---

图7-19 CDI示例首页

7.7.2 引擎配置与流程定义

为了演示方便本节示例使用JBoss中用于演示的内置数据源（内存模式的H2数据库）及容器提供的事务管理器。下面列出了本节示例的Activiti配置文件（activiti.cfg.xml）。

```
<!-- JTA事务管理器 -->
<bean id="transactionManager"
      class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName" // JBoss内置的事务管理器
    value="java:jboss/TransactionManager"/ >
  <property name="resourceRef" value="true" />
</bean>
```

```
<!-- 流程引擎配置对象 -->
<bean id="processEngineConfiguration"
      class="org.activiti.cdi.
        CdiJtaProcessEngineConfiguration">
  <property name="dataSourceJndiName"
    value="java:jboss/datasources/ExampleDS" /> // 内置的数据源
  <property name="databaseType" value="h2" />
  <property name="transactionManager" ref="transactionManager" />
  <!-- 打开外部事务管理器开关-->
  <property name="transactionsExternallyManaged"
    value="true" />
  <property name="databaseSchemaUpdate" value="true" />
</bean>
```

对于以上配置有几点需要说明，

- **引擎配置对象**：使用activiti-cdi模块的org.activiti.cdi.CdiJtaProcessEngineConfiguration。
- **数据源**：通过JNDI方式从容器获取。
- **事务管理器**：通过JNDI方式从容器获取JTA事务管理器（分布式事务管理），“java:jboss/TransactionManager”为JBoss内置的事务管理器，并且设置引擎的“transactions-ExternallyManaged”属性为“true”，告知引擎事务由“transactionManager”属性引用的bean作为引擎的事务管理器。

Activiti CDI模块与Activiti Spring模块有一个类似的功能——自动部署流程：在源码的资源根目录中添加名称为“processes.xml”的文件并在该文件中加入需要部署的流程文件路径，在启动引擎并初始化Activiti引擎时会自动把该文件中定义的流程部署到引擎。相关配置代码如下：

```
<?xml version="1.0" encoding="utf-8" ?>
<!-- 自动部署的流程文件 -->
<processes>
  <process resource="diagrams/leave.bpmn" />
</processes>
```

所以，在启动chapter7-cdi示例后单击“流程定义”菜单就可以查询到如图7-20所示的页面，其中显示了已经部署到引擎的流程定义列表。不过需要注意的是每次启动应用时引擎都会部署processes.xml文件中配置的流程文件，所以建议在生产环境选择手动部署而不是由CDI模块自动部署。



图7-20 流程定义列表

7.7.3 流程定义与启动

本节的示例流程定义使用了外值表单模式，在启动节点以及每个用户任务上用activiti:formKey属性定义流程的表单路径，相关配置如下：

```
<process id="leave" name="请假流程-CDI">
  <startEvent id="startevent1" name="Start"
    activiti:initiator="applyUserId"
    activiti:formKey="leave-apply.xhtml"/>
  <userTask id="deptLeaderAudit" name="部门领导审批"
    activiti:assignee="deptLeader"
    activiti:formKey="form/deptLeaderAudit.xhtml"/>
  <userTask id="modifyApply" name="调整申请"
    activiti:assignee="${applyUserId}"
    activiti:formKey="form/modifyApply.xhtml" />
  <userTask id="hrAudit" name="人事审批"
    activiti:assignee="hr"
    activiti:formKey="form/hrAudit.xhtml" />
  <userTask id="reportBack" name="销假"
    activiti:assignee="${applyUserId}"
    activiti:formKey="form/reportBack.xhtml"/>
</process>
```

以上流程定义片段中加粗部分为任务办理人以及任务表单URL，对于流程中的排他网关的条件定义也与前面章节介绍的有些不同，例如判断部门领导是否同意的条件为：

```
 ${!leave.isDeptLeaderApproved() }
```

其中leave为一个普通的Pojo对象，该对象的属性如下，特别的是在该类上标注了两个注解，@Named注解可以通过leave获取到该类的实例，@BusinessProcessScoped注解则由Activiti CDI模块提供，功能类似JAVA EE的@ConversationScoped注解，可以把该对象绑定到一次请求中。

```
@Named
@BusinessProcessScoped
public class Leave implements Serializable {
    // 业务属性
    private String startTime;
    private String endTime;
    private String reason;
    private String reallyStartTime;
    private String reallyEndTime;
    // 网关条件判断属性
    private boolean reapply;
    private boolean deptLeaderApproved;
    private boolean hrApproved;
    // ...
}
```

读取流程定义的方式与以往没有差别，只不过使用了JAVA EE的相关技术而已（例如注入、JSF等）。下面的代码清单列出了ActivitiAction类中读取流程定义的代码块。

```
@Inject // 注入引擎接口
private RepositoryService repositoryService;
@Produces
@Named("processDefinitionList")
public List<ProcessDefinition> getProcessDefinitionList() {
    return repositoryService
        .createProcessDefinitionQuery().list();
}
```

上面的方法被定义成一个Bean，可以通过名称“processDefinitionList”调用，在视图文件“processList.xhtml”中就可以循环流程定义集合了（结果见图7-20）。相关代码如下：

```
<h:dataTable value="#{processDefinitionList}"
    var="process">
    // 省略了其他几列
    <h:column>
    <f:facet name="header">操作</f:facet>
    <h:outputLink value="#{formService
        .getStartFormData(process.id).formKey}">
        启动
    <f:param name="processDefinitionKey"
        value="#{process.key}"></f:param>
    </h:outputLink>
    </h:column>
</h:dataTable>
```

上面代码清单中“h:outputLink”部分的加粗代码是重点，从Activiti的FormService接口中获取启动流程的formKey属性，从而获取流程文件中startEvent标签的“activiti:formKey”属性定义的值，即“leave-apply.xhtml”；在单击图7-20的“启动”链接后页面跳转到如图7-21所示的请假申请界面。

图7-21 请假申请界面

在请假申请文件leave-apply.xhtml中把输入框的值绑定到Leave对象的属性上，这样当单击图7-21的“申请”按钮后容器会把界面填写的值写回到Leave对象，有关界面的定义如下：

```
<tr>
    <td>申请人:</td>
    <td>
        <h:inputText value="#{systemUser.userId}" />
    </td>
</tr>
// http://www.hzcourse.com/resource/readBook?path=/openresources/teach_ebook/uncompressed/15030/OEBPS/Text/... 省略了其他的业务字段定义
<tr>
    <td></td>
    <td><h:commandButton value="申请"
        action="#{activitiCdiAction.startLeaveProcess('102')}"
        /></td>
</tr>
```

以上代码中的systemUser是一个简单的Bean对象，仅提供一个userId属性来模拟真实业务系统中的当前用户。

“申请”按钮的“action”属性为当前页面中表单的action，作用是启动一个流程，请求由名称为“activitiCdiAction”的startLeaveProcess方法处理，相关代码如下：

```
@Named
public class ActivitiCdiAction {
    @ProcessVariable String orgId;
    @Inject
    IdentityService identityService;
    @Inject
    private SystemUser systemUser;
    @StartProcess(value = "leave")
    public String startLeaveProcess(String orgId) {
        identityService
            .setAuthenticatedUserId(systemUser.getUserId());
        this.orgId = orgId;
        System.out.println("流程【请假】启动" + orgId);
        return "taskList.xhtml";
    }
}
```

在ActivitiCdiAction中分别注入了引擎的IdentityService接口及当前用户对象，标注了@ProcessVariable注解的属性则表示在流程启动时需要保存的变量。从上面的代码清单中可以看出方法startLeaveProcess接口的orgId参数赋值给了成员变量orgId，该方法上标注的@StartProcess注解用来声明当该方法被调用时启动一个KEY为“leave”的流程，同时扫描该类中所有标注了@ProcessVariable注解的属性作为流程变量。因此单击了图7-21中的“申请”后就成功启动了一个请假流程，可以在待办任务以及流程变量菜单中验证。

图7-22展示了启动流程后的流程变量列表。

流程变量列表

名称	流程实例ID	任务ID	类型	值
leave	4		serializable	me.kafeitu.activiti.chapter7.cdi.bean.Leave@43bdb9f3
applyUserId	4		string	kafeitu
orgId	4		string	102

图7-22 流程变量列表

从图7-22中可以看出，在启动流程后引擎保存了三个变量，其中applyUserId为申请人ID，orgId为部门ID（为了演示这里固定使用102作为部门ID），名称为“leave”的变量类型为“serializable”，是一个对象类型。正因为Leave类被标注了@BusinessProcessScoped注解所以才会作为变量保存到引擎中。

需要注意的是在使用@StartProcess注解之前要在WEB-INF/beans.xml文件中添加处理该注解的拦截器StartProcessInterceptor，该拦截器由Activiti CDI模块提供，配置拦截器的相关配置如下，同时也包含了@CompleteTask的注解拦截器（稍后会介绍）。

```
<interceptors>
  <class>org.activiti.cdi.impl.annotation
    .StartProcessInterceptor</class>
  <class>org.activiti.cdi.impl.annotation
    .CompleteTaskInterceptor</class>
</interceptors>
```

StartProcessInterceptor拦截器会判断方法上是否被标注了@StartProcess注解，如果被标注了，则读取value属性的值，然后调用Activiti CDI模块中BusinessProcess类的startProcessByKey方法启动流程。也可以采用在业务处理层自行注入BusinessProcess对象调用响应的方法启动流程。

启动流程的方式除了使用@StartProcess注解外还可以使用调用BusinessProcess的方法，例如，图7-21中“申请”的action可以这样写：

```
# {businessProcess.startProcessByKey (processDefinitionKey) }
```

也可以正常启动流程，只不过这种方式不能传递orgId变量，所以一般情况下不会使用这种方式。

7.7.4 任务办理与完成

启动流程后任务到达“部门领导审批”节点，该节点的办理人为“deptLeader”（为了演示方便固定了该任务的办理人），单击“待办任务”菜单项，在输入框中输入“deptLeader”后页面立即读取该用户的待办任务列表，如图7-23所示。



图7-23 deptLeader用户的待办任务列表

在taskList.xhtml文件中输出任务列表的部分代码如下：

```
<h:dataTable value="#{todoTaskList}" var="v_task">
  http://www.hzcourse.com/resource/readBook?path=/openresources/teach_ebook/uncompressed/15030/OEBPS/Text/...
  <h:column>
    <f:facet name="header">操作</f:facet>
    <h:outputLink value="#{request.contextPath}
      /#{formService.getTaskFormData(v_task.id).formKey}">
      办理
      <f:param name="taskId"
        value="#{v_task.id}"></f:param>
    </h:outputLink>
  </h:column>
</h:dataTable>
```

在以上代码中，“办理”链接的读取方式与启动流程类似，获取任务的formKey属性并与上下文路径拼接，所以在点击“办理”链接后页面跳转到了<http://localhost:8080/chapter7-cdi/form/deptLeaderAudit.xhtml?taskId=13>页面，如图7-24所示。

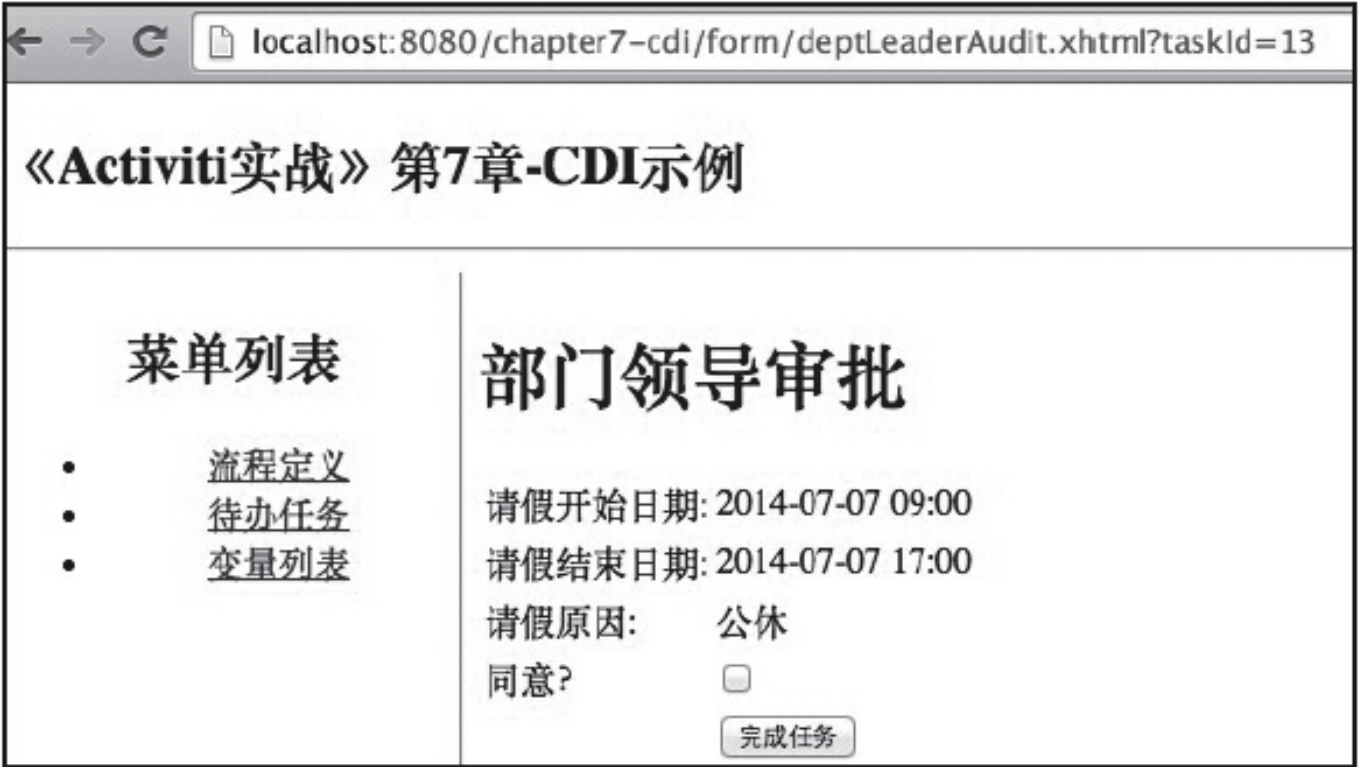


图7-24 任务办理界面

查看本节源码的form/deptLeaderAudit.xhtml文件可以看到如下的代码片段：


```
<f:metadata>
  <f:viewParam name="taskId" />
  <!-- 办理任务时读取Task对象并开始一个新的会话 -->
  <f:event type="preRenderView"
    listeners="#{businessProcess.startTask(taskId, true)}" />
</f:metadata>
<h1>#{task.name}</h1>
<h:form>
  <table>
    // 省略了部门tr输出
    <tr>
      <td>同意? </td>
      <td>
        <h:selectBooleanCheckbox // 把结果保存到leave
          value="#{leave.deptLeaderApproved}" /></td>
    </tr>
    <tr>
      <td>
        <h:commandButton value="完成任务"
          action="#{activitiCdiAction.completeTask()}" />
      </td>
    </tr>
  </table>
</h:form>
```

以上代码分为两部分，第一部分（<f:metadata>）获取视图的taskId参数，然后用event标签定义一个页面来展示之前的监听器，调用Activiti CDI模块BusinessProcess类的startTask方法，该方法可以从引擎读取一个任务对象（Task），该方法的第二个参数为true，作用是让容器开启一个会话（Conversation），这样可以把读取到的Task对象绑定到本次会话中，当然流程变量也会被读取到且可以直接调用。

第二部分（<h:form>）则用来展示业务信息与“完成任务”按钮，其中“同意？”选项的值被绑定到leave对象的deptLeaderApproved属性上（Boolean类型）；单击“完成任务”按钮会调用ActivitiCdiAction类的completeTask方法从而触发任务完成动作，任务完成和启动流程类似，Activiti CDI模块也提供了注解，名称为@CompleteTask，使用该注解定义的方法如下：

```
@CompleteTask(endConversation = true)
public String completeTask() {
    return "/taskList.xhtml";
}
```

以上代码中的completeTask方法被标注了@CompleteTask注解，并且设置属性endConversation为true，表示在任务完成后本次会话是否结束，在实际应用中可以根据业务的不同决定在任务完成后是否要结束本次会话。

当然除了使用注解声明方式完成任务外，还可以调用BusinessProcess对象的相应方法，在“人事审批”节点的页面文件hrAudit.xhtml中有如下的代码，同样也可以完成任务。

```
<h:commandButton value="完成任务"
  action="#{businessProcess.completeTask(true)}" />
```

7.7.5 事件监听

Activiti CDI模块除了提供启动流程、完成任务之外还提供了事件监听方面的注解，运行用户自定义事件处理器，并且支持条件过滤，例如监听任务“部门领导审批”的创建事件可以用以下代码定义事件处理器。

```
public class ActivitiCdiEventListener {
    public void onStartActivityService(
        @Observes @CreateTask("deptLeaderAudit")
        BusinessProcessEvent businessProcessEvent) {
        System.out.println("捕获到【部门领导审批】的事件: "
            + businessProcessEvent);
    }
}
```

不过仅仅定义好事件处理器还不够，还需要在引擎配置文件中添加针对CDI的事件处理器，该处理器由Activiti CDI模块提供，相关配置如下：

```
<property name="postBpmnParseHandlers">
  <list>
    <bean class="org.activiti.cdi.impl.event
      .CdiEventSupportBpmnParseHandler" />
  </list>
</property>
```

在启动流程后在JBoss控制台就可以看到如下的输出内容：

```
[stdout] (http--127.0.0.1-8080-1) 捕获到【部门领导审批】开始事件: Event 'leave' ['create', deptLeaderAudit]
```

Activiti CDI模块提供了几种常用的事件注解，具体如下：

- BusinessProcess，启动流程
- CreateTask，任务创建
- AssignTask，任务分配办理人
- CompleteTask，任务完成
- DeleteTask，删除任务
- StartActivity，开始一个活动
- EndActivity，结束一个活动
- TakeTransition，Flow被执行时

7.8 本章小结

本章主要讲解了Activiti与各种不同的容器的集成，分别介绍了基于Spring的XML配置方式、基于Spring注解配置方式、基于Spring Boot的注解方式、基于JAVA EE CDI的方式。这几种集成方式各不相同，也有着不同的应用场景。

在讲解与Spring集成时主要介绍了：引擎工厂、表达式、监听器，以及综合运用。其中引擎工厂、表达式、监听器先从默认支持讲解，再讲解如何使用Spring代理实现。Activiti提供了多种创建引擎的方式，本章对不同的创建方式做了简单介绍，对使用Spring方式创建引擎做了详细的介绍，因为毕竟目前大多数企业应用都是基于Spring的。

表达式与监听器是Activiti中比较重要的一部分内容，所以用了两节详细讲解两者如何使用，以及如何联合起来使用，例如通过监听器执行一个表达式。

7.5节综合了前面所讲内容实现了普通表单形式的请假流程，既补充了对三种表单（普通表单、动态表单、外置表单）中普通表单的介绍，又针对Spring如何做到统一事务管理做了详细介绍，这也是为什么把普通表单在本章介绍而非第6章介绍的原因。

除了传统的基于XML的方式配置、初始化引擎的方式之外还可以通过注解的方式由Spring、Spring Boot、CDI等容器来扫描指定注解，以实现初始化或其他与流程相关的功能。

第8章 邮件服务

邮件系统可以说是每个企业办公系统中必备的，利用现有的邮件资源把邮件任务加入到流程中作为其中一个环节，例如，在一个任务创建或超时之后发送邮件通知员工办理（如果企业邮箱支持PUSH功能就可以即时提醒）；在一些客服系统中会由客服处理流程，对于单次的流程办理会通过邮件通知客户，可以根据业务信息及预设的模板自动创建邮件内容并由流程引擎发送通知邮件给最终客户。

本章首先介绍Activiti提供的邮件任务（4.3.6节）的使用方法，再把邮件任务加入到第7章的请假流程中，在审批通过之后发送通知邮件，如果“销假”节点超过限制时间，同样发送邮件通知用户。

8.1 配置与测试

8.1.1 搭建邮件系统

邮件系统有太多可以选择了，有商业的、开源的，例如企业常用的Postfix、zimbra、james等。为了方便讲解，本章使用Apache的James（Java Apache Mail Enterprise Server）作为邮件系统，它是Apache的子项目之一，完全采用Java实现，支持SMTP\POP3等多种协议，还可以自行扩展。可以从<http://james.apache.org/>下载James。这里以James稳定版（版本号为2.3.2）为例来介绍邮件系统的搭建。

James的安装方法也比较简单，把下载的压缩包解压得到类似图8-1的目录结构。

▼	folder	james-2.3.2	Today 2:14 PM
▶	folder	bin	Aug 10, 2009 4:51 PM
▶	folder	conf	Aug 10, 2009 4:51 PM
▶	folder	ext	Aug 10, 2009 4:51 PM
▶	folder	lib	Aug 10, 2009 4:51 PM
▶	folder	tools	Aug 10, 2009 4:51 PM
▶	folder	apps	Today 2:14 PM
▶	folder	temp	Today 5:27 PM
▶	folder	logs	Today 8:14 PM
▶	folder	work	Today 8:14 PM

图8-1 James的目录结构

启动James可以通过执行bin目录的run.sh（Linux或者UNIX系统）和run.bat（Windows系统）脚本来完成，不过在运行之前需要修改几个端口号的参数，因为有些操作系统在启动James时会提示SMTP或其他协议端口分配权限不足，所以需要修改James的端口号：打开james_home/apps/james/SAR-INF/config.xml文件，修改该文件中的三个端口号（pop3server、smtpserver、nntpserver中port标签的值），保存文件后重新启动James服务。

```
<pop3server enabled="true">
  <port>110</port>
</pop3server>
<smtpserver enabled="true">
  <port>25</port>
</smtpserver>
<nntpserver enabled="true">
  <port>119</port>
</nntpserver>
```

为了满足后面的测试需求，这里把smtp协议的端口号改为2025。

通过执行bin目录的run脚本显示如图8-2的信息，表示服务启动成功。


```
➔root@hy-mbp /Users/henryyan/work/sources/apache/james/james-2.3.2/bin ./run.sh
Using PHOENIX_HOME: /Users/henryyan/work/sources/apache/james/james-2.3.2
Using PHOENIX_TMPDIR: /Users/henryyan/work/sources/apache/james/james-2.3.2/temp
Using JAVA_HOME: /Library/Java/JavaVirtualMachines/1.7.0.jdk/Contents/Home
Running Phoenix:

Phoenix 4.2

James Mail Server 2.3.2
Remote Manager Service started plain:4555
POP3 Service started plain:110
SMTP Service started plain:2025
NNTP Service started plain:119
FetchMail Disabled
```

图8-2 James启动成功

James默认使用localhost作为域名，如果需要更改，可以在config.xml文件中修改servername标签的值。

在图8-2中提示了“Remote Manager Service”，意思是可以通过telnet协议登录James管理邮件系统，默认的用户名和密码为：root和root。如图8-3所示，成功登录到了James后台。

```
➔henryyan@hy-mbp ~ telnet localhost 4555
Trying ::1...
Connected to localhost.
Escape character is '^]'.
JAMES Remote Administration Tool 2.3.2
Please enter your login and password
Login id:
root
Password:
root
Welcome root. HELP for a list of commands
help
Currently implemented commands:
help                display this help
listusers           display existing accounts
countusers          display the number of existing accounts
adduser [username] [password] add a new user
verify [username]   verify if specified user exist
deluser [username]  delete existing user
setpassword [username] [password] sets a user's password
setalias [user] [alias] locally forwards all email for 'user' to 'alias'
showalias [username] shows a user's current email alias
unsetalias [user]   unsets an alias for 'user'
setforwarding [username] [emailaddress] forwards a user's email to another email address
showforwarding [username] shows a user's current email forwarding
unsetforwarding [username] removes a forward
user [repositoryname] change to another user repository
shutdown            kills the current JVM (convenient when James is run as a daemon)
quit                close connection
```

图8-3 通过telnet成功登录了James的后台

接下来通过adduser命令为邮件系统添加用户。如图8-4所示，添加了用户tom，并设置密码为000000，通过listusers命令可以列出当前的用户列表。

```
adduser tom 000000
```

```
➔henryyan@hy-mbp ~ telnet localhost 4555
Trying ::1...
Connected to localhost.
Escape character is '^]'.
JAMES Remote Administration Tool 2.3.2
Please enter your login and password
Login id:
root
Password:
root
Welcome root. HELP for a list of commands
adduser tom 000000
User tom added
listusers
Existing accounts 1
user: tom
```

图8-4 添加邮件系统用户tom

接下来利用邮件客户端验证邮件功能是否正常，图8-5、图8-6、图8-7和图8-8分别展示了利用邮件客户端添加tom@localhost账号的步骤。

Add Account

You'll be guided through the steps to set up an additional account.

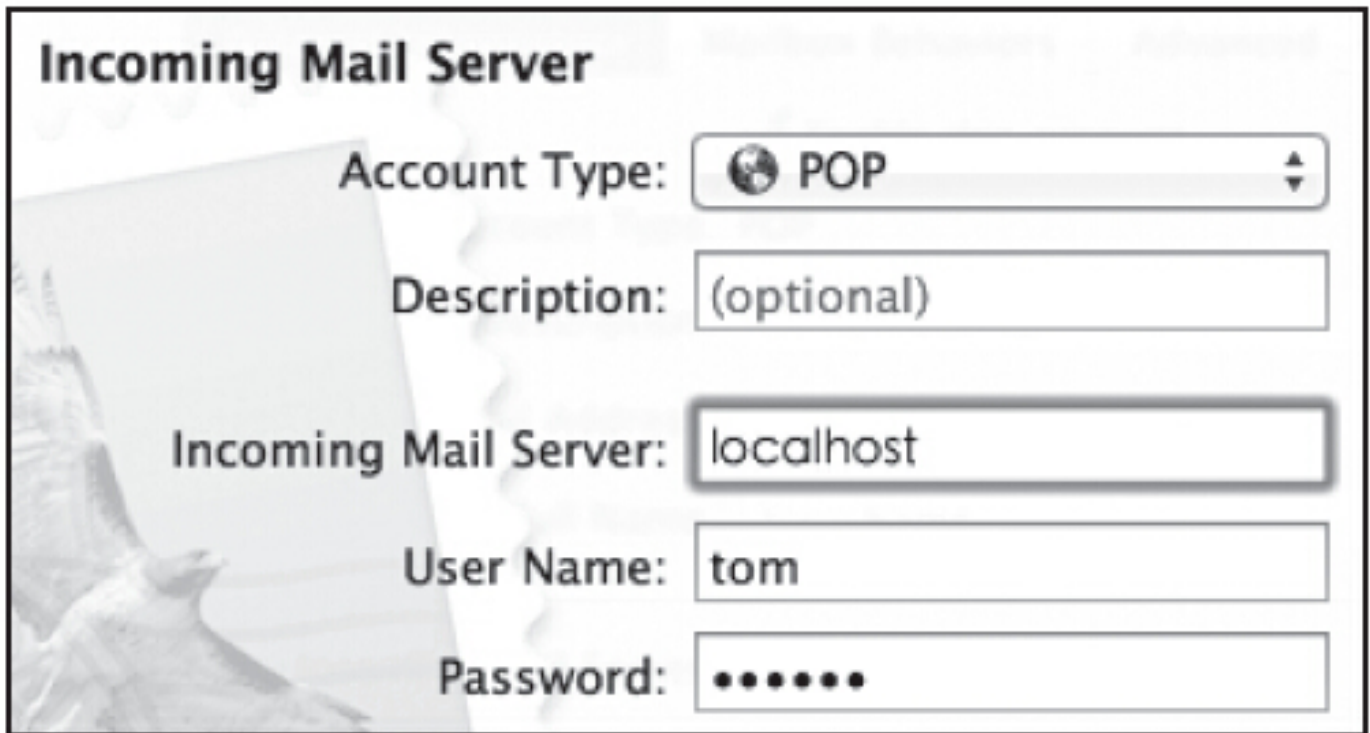
To get started, provide the following information:

Full Name:

Email Address:

Password:

图8-5 添加邮件账号



Incoming Mail Server

Account Type:

Description:

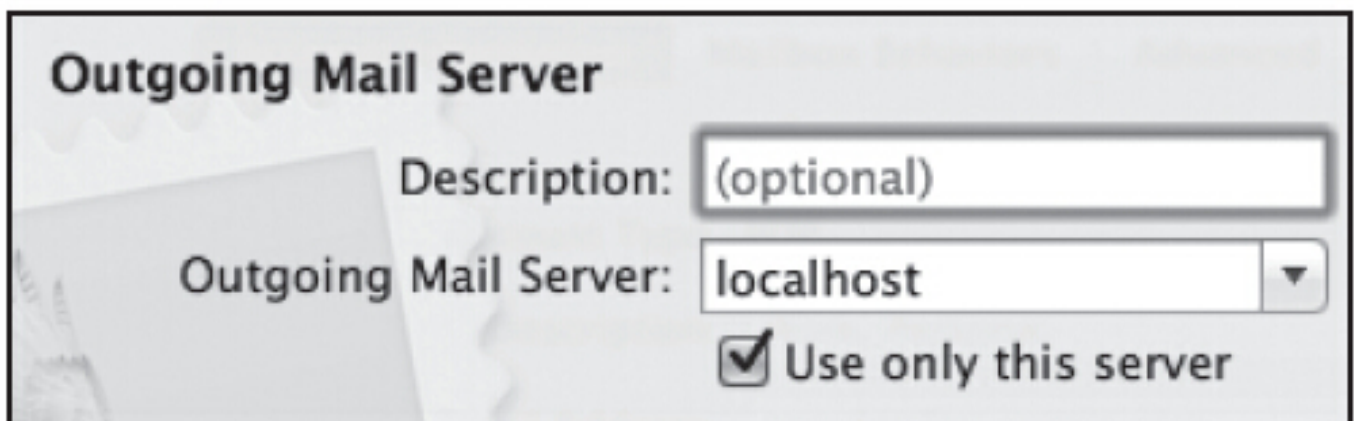
Incoming Mail Server:

User Name:

Password:

The image shows a window titled 'Incoming Mail Server' with a background of a mountain landscape. It contains five input fields: 'Account Type' (a dropdown menu showing 'POP'), 'Description' (a text box with '(optional)'), 'Incoming Mail Server' (a text box with 'localhost'), 'User Name' (a text box with 'tom'), and 'Password' (a password box with five dots).

图8-6 设置POP协议



Outgoing Mail Server

Description:

Outgoing Mail Server:

☒ Use only this server

The image shows a window titled 'Outgoing Mail Server' with a background of a mountain landscape. It contains three input fields: 'Description' (a text box with '(optional)'), 'Outgoing Mail Server' (a text box with 'localhost'), and a checkbox labeled 'Use only this server' which is checked.

图8-7 设置SMTP服务器

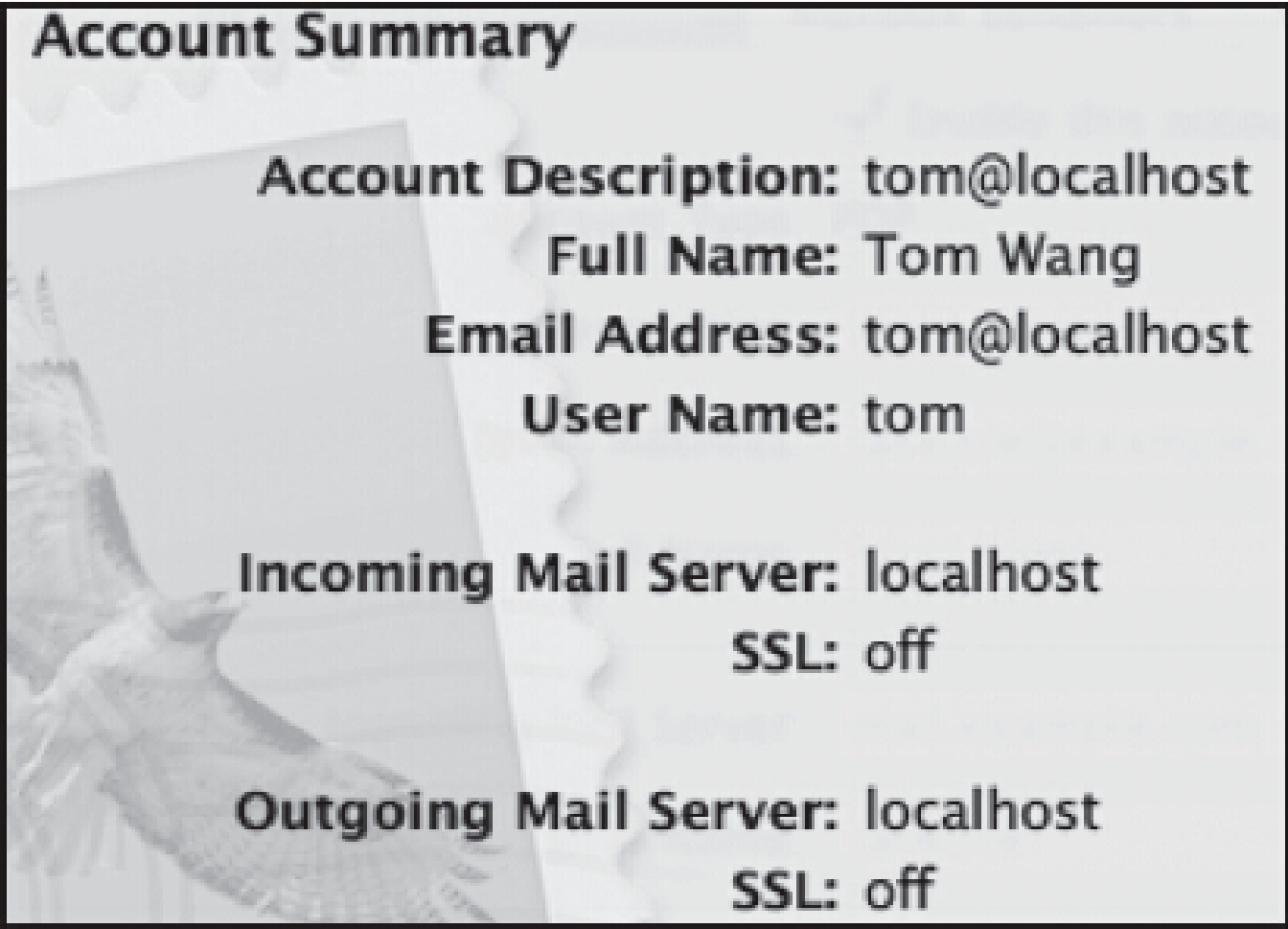


图8-8 邮件账号信息汇总

给自己发送一封邮件来验证是否可以正常的发送（如图8-9所示）与接收邮件（如图8-10所示）。

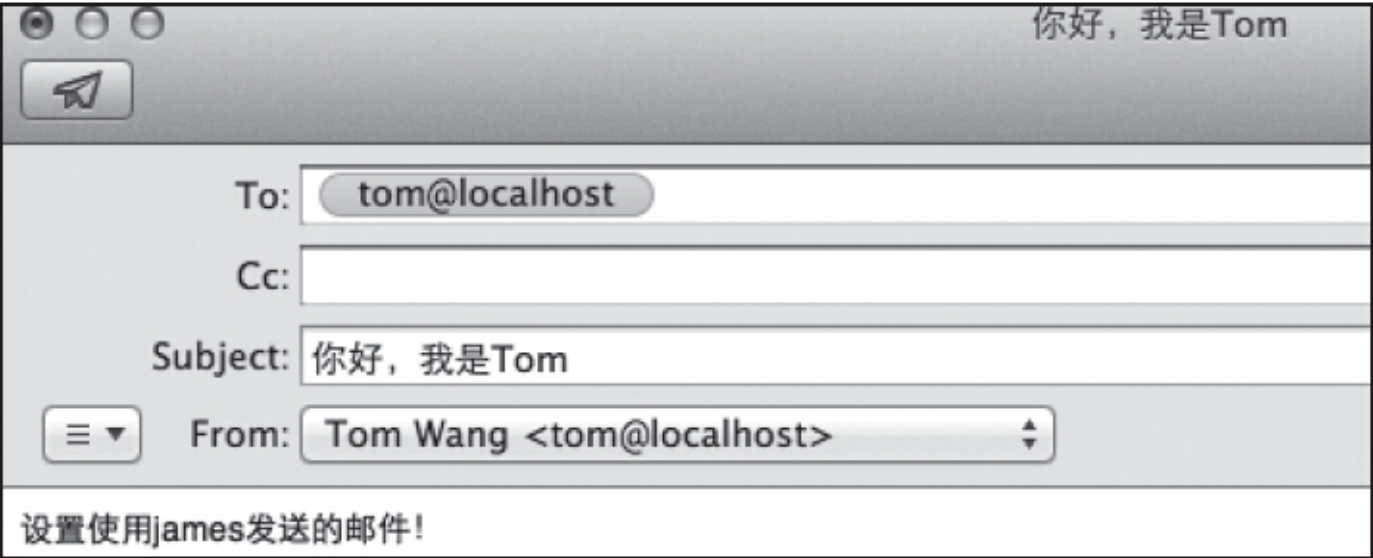


图8-9 发送邮件给tom@localhost



图8-10 接收到新邮件

回归主题，邮件系统搭建好了，如何在流程中通过Activiti的邮件任务自动发送邮件呢？在4.3.6节中详细介绍了和邮件有关的配置项，下面通过只包含一个邮件任务的流程（如图8-11所示）来了解自动发送邮件的配置，对应的代码如代码清单8-1所示。



图8-11 只包含一个邮件任务的流程

代码清单8-1 图8-11对应的XML

```
<process id="testMailTask" name="My process" isExecutable="true">
  <startEvent id="startevent1" name="Start"></startEvent>
  <serviceTask id="mailtask1" name="Mail Task" activiti:type="mail">
    <extensionElements>
      <activiti:field name="to"> // 邮件接收方地址
      <activiti:expression>${to}</activiti:expression>
      </activiti:field>
      <activiti:field name="from"> // 邮件发送方地址
      <activiti:expression>${from}</activiti:expression>
      </activiti:field>
      <activiti:field name="subject">
      <activiti:expression>你好, ${name}</activiti:expression>
      </activiti:field>
      <activiti:field name="charset">
      <activiti:string>UTF-8</activiti:string>
      </activiti:field>
      <activiti:field name="html">
      <activiti:expression>
        <![CDATA[ // 邮件内容
          <html>
            <body>
              你好 ${name},<br/><br/>
              如果你收到这封邮件，恭喜你已经学会了如何通过Activiti的邮件任务发送邮件！
            </body>
          </html>
        ]]>
      </activiti:expression>
      </activiti:field>
    </extensionElements>
  </serviceTask>
  <sequenceFlow id="flow1" sourceRef="startevent1" targetRef="mailtask1"></sequenceFlow>
  <endEvent id="endevent1" name="End"></endEvent>
  <sequenceFlow id="flow2" sourceRef="mailtask1" targetRef="endevent1"></sequenceFlow>
</process>
```

对应地，通过一个简单的单元测试启动流程从而触发邮件任务的执行（该流程只有一个任务），单元测试代码见代码清单8-2。

代码清单8-2 测试邮件任务——James作为邮件服务器

```
public class MailTaskTest { // 未继承AbstractTest
  @Rule // 使用一个单独的activiti-mail.cfg.xml作为引擎配置
  public ActivitiRule activitiRule = new ActivitiRule("activiti-mail.cfg.xml");
  @Test
  @Deployment(resources = { "diagrams/chapter8/testMailTask.bpmn" })
  public void sendMailLocalTest() throws Exception {
    Map<String, Object> variables = new HashMap<String, Object>();
    variables.put("name", "Tom Wang"); // 用来替换邮件中的${name}
    variables.put("to", "tom@localhost");
    variables.put("from", "noreply@localhost");
    ProcessInstance processInstance = activitiRule.getRuntimeService()
      .startProcessInstanceByKey("testMailTask", variables);
    assertNotNull(processInstance);
  }
}
```

在代码清单8-2中使用activiti-mail.cfg.xml文件创建了一个ActivitiRule的对象，在此文件中配置了邮件服务的参数，见代码清单8-3所示。

代码清单8-3 引擎配置文件activiti-mail.cfg.xml

```
<bean id="processEngineConfiguration" class="org.activiti.engine.impl.cfg.StandaloneInMemProcessEngineConfiguration">
  <property name="databaseSchemaUpdate" value="true"/>
  <property name="mailServerPort" value="2025"/>
</bean>
```

代码清单8-3覆盖了参数mailServerPort的默认值（默认为25），使之和搭建的James参数保持一致。

在运行了代码清单8-2的单元测试之后，刷新邮件客户端的列表就可以看到如图8-12所示的新邮件。

noreply@localhost3:29 PM 你好，Tom Wang 你好 Tom Wang, 如果你收到这封邮件，恭喜你已经学会了如何通过Acti... Tom Wang12:52 AM 你好，我是Tom 设置使用james发送的邮件！	noreply@localhost To: tom@localhost 你好，Tom Wang 你好 Tom Wang, 如果你收到这封邮件，恭喜你已经学会了如何通过Activiti的邮件任务发送邮件！
---	---

图8-12 运行代码清单8-2之后收到的新邮件

8.1.3 使用Gmail发送邮件

Gmail的SMTP使用SSL加密协议，这和大多数企业邮箱一样，为了安全起见把邮件内容加密。不过在通过Activiti发送邮件时需要配置引擎的mailServerUseSSL属性为“true”。

代码清单8-4列出了使用Gmail作为邮件服务器的配置。

代码清单8-4 在引擎配置中设置Gmail的参数

```
<bean id="processEngineConfiguration" class="org.activiti.engine.impl.cfg.StandaloneInMemProcessEngineConfiguration">
  <property name="databaseSchemaUpdate" value="true"/>
  <property name="mailServerHost" value="smtp.gmail.com" />
  <property name="mailServerUseSSL" value="true" /> // 启用加密
  <property name="mailServerPort" value="465" />
  <property name="mailServerUsername" value="yanhonglei@gmail.com" />
  <property name="mailServerPassword" value="password" />
</bean>
```

代码清单8-4覆盖了邮件配置的默认参数，并且启用SSL。读者可以把mailServerUsername和mailServerPassword参数更改为自己的Gmail账号运行MailTaskByGmailTest（见代码清单8-5）。运行之后即可收到邮件，如图8-13所示。

代码清单8-5 使用Gmail作为邮件服务器

```
public class MailTaskByGmailTest {
    @Rule
    public ActivitiRule activitiRule = new ActivitiRule("activiti-gmail.cfg.xml");
    @Test
    @Deployment(resources = { "diagrams/chapter8/testMailTask.bpmn" })
    public void sendMailLocalTest() throws Exception {
        Map<String, Object> variables = new HashMap<String, Object>();
        variables.put("name", "Tom Wang");
        variables.put("to", "yanhonglei@gmail.com");
        variables.put("from", "yanhonglei@gmail.com");
        ProcessInstance processInstance = activitiRule.getRuntimeService()
            .startProcessInstanceByKey("testMailTask", variables);
        assertNotNull(processInstance);
    }
}
```

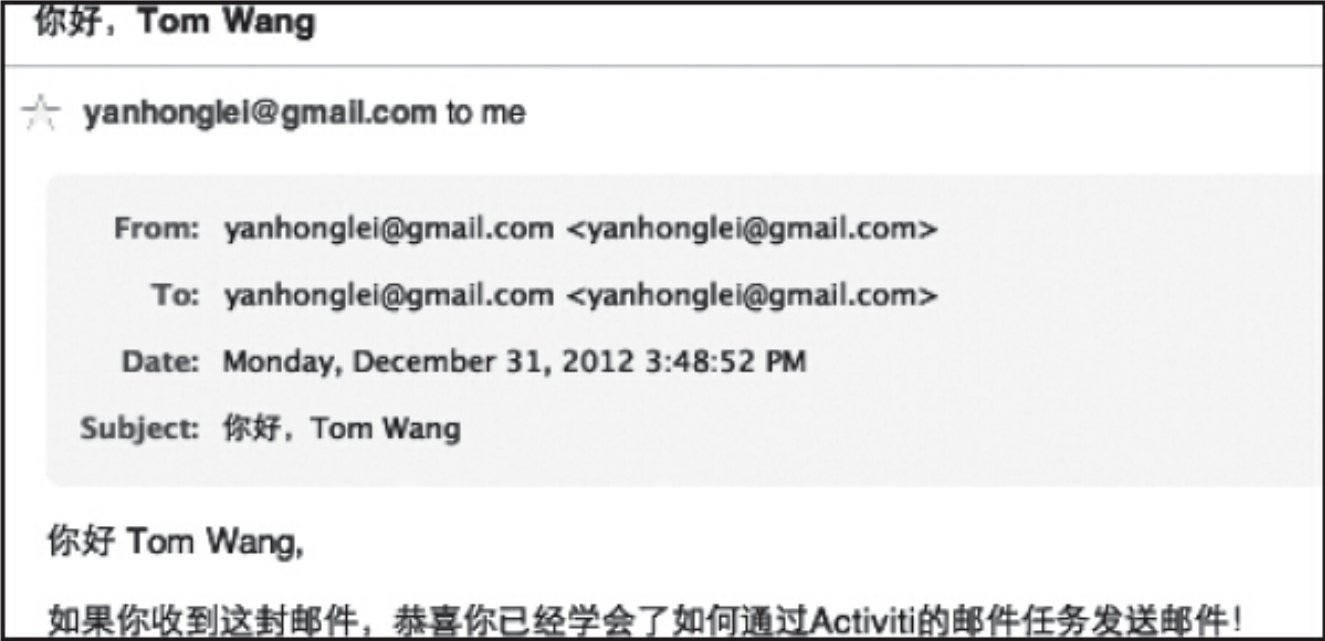


图8-13 收到了通过Gmail发送的邮件

8.2 与业务集成

8.2.1 即时发送邮件

在第6章和7.5节中分别使用三种表单方式实现了请假流程的办理，本节将在6.1节流程定义的基础上继续扩充，为流程添加发送邮件功能，这样可以在审批通过之后发送通知邮件。

添加了邮件任务的流程图如图8-14所示。

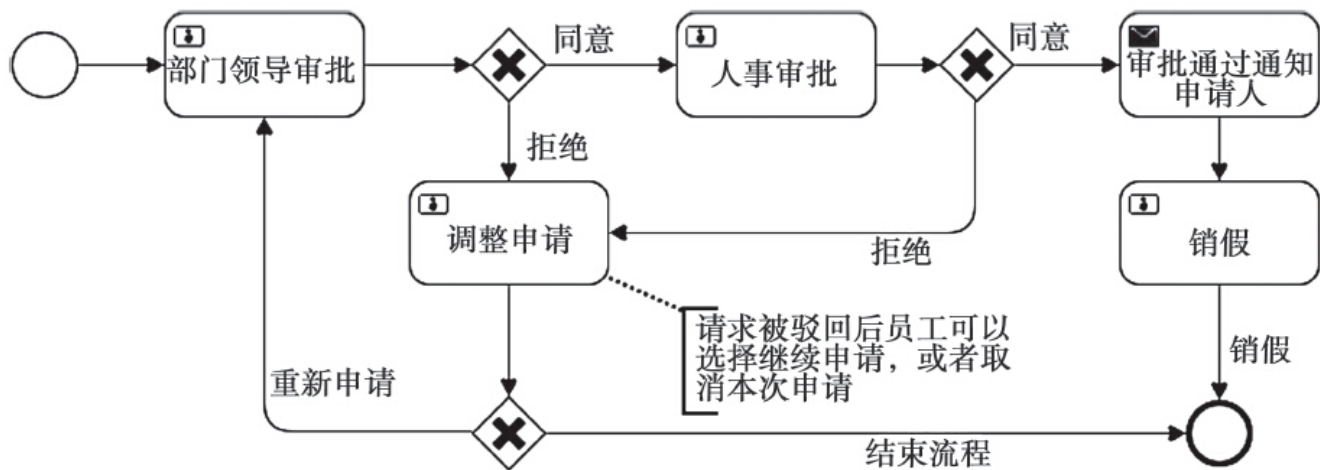


图8-14 请假流程——在审批通过之后发送通知邮件

由于6.1节中已经详细讲解了该请假流程的XML定义的含义，因此代码清单8-6只列出了新添加的邮件任务“审批通过通知申请人”的XML代码。

代码清单8-6 邮件任务XML配置

```
<serviceTask id="sendMailForApproved" name="审批通过通知申请人" activiti:type="mail">
  <extensionElements>
    <activiti:field name="to">
      <activiti:expression>${to}</activiti:expression>
    </activiti:field>
    <activiti:field name="from">
      <activiti:string>noreply@localhost</activiti:string>
    </activiti:field>
    <activiti:field name="subject">
      <activiti:string>请假申请已审批通过</activiti:string>
    </activiti:field>
    <activiti:field name="charset">
      <activiti:string>UTF-8</activiti:string>
    </activiti:field>
    <activiti:field name="html">
      <activiti:expression><![CDATA[
        <html>
          <body>
            你好 ${name},<br/><br/>          // ${name}用于获取申请人的姓名
            <p>您申请的请假已经审批通过。</p>
            <hr/>
            <div>开始时间: ${startDate}</div>          // 请假的信息取自表单属性
            <div>结束时间: ${endDate}</div>
            <div>请假原因: ${reason}</div>
          </body>
        </html>
      ]]>
    </activiti:field>
    <activiti:executionListener event="start" class="me.kafeitu.activiti.chapter8.SetMailInfo"/>
  </extensionElements>
</serviceTask>
```

其中，在邮件任务被引擎创建时添加了监听“start”事件的监听器，该监听器可以在执行任务之前将发送邮件的一些信息设置到变量中，以便从变量中获取邮件主题信息。代码清单8-7列出了Start监听器的代码。

代码清单8-7 邮件任务Start监听器——设置发送邮件时的一些变量

```
public class SetMailInfo implements ExecutionListener {
  public void notify(DelegateExecution execution) throws Exception {
    // 从Activiti 5.11版本开始可以通过DelegateExecution对象获取引擎的7个Service对象实例
    IdentityService identityService = execution.getEngineServices().getIdentityService();
    String applyUserId = (String) execution.getVariable("applyUserId"); // 获取申请人ID
    User user = identityService.createUserQuery().userId(applyUserId).singleResult();
    execution.setVariableLocal("to", user.getEmail()); // 申请人email
    execution.setVariableLocal("name", user.getFirstName() + " " + user.getLastName());
  }
}
```

最后将leave-mail.bpmn和leave-mail.png打包成Zip文件后部署，先用账号tom登录申请，然后分别用kermit和jeeny完成“部门领导审批”和“人事审批”任务办理，当“人事审批”任务完成时刷新邮件列表即可收到一封新邮件，如图8-15所示。



图8-15 通过Activiti的邮件任务发送的邮件——请假申请已审批通过

8.2.2 定时发送邮件

除了8.2.1的即时发送邮件，在日常的应用中还有另外一个需求——定时发送邮件，原因有多种，例如考虑服务器压力（晚上批量处理邮件队列）、过期提醒等。

假设现在要为请假流程添加一个功能，如果用户在申请的假期结束后1天之内没有“销假”，则发送通知邮件。这个功能就要利用4.6.1小节介绍的定时器边界事件来完成，在“销假”节点添加一个定时器边界事件，以“请假结束时间”为基数加上一天作为定时发送邮件的时间，在触发了定时器边界事件之后执行输出流，也就是邮件任务。图8-16展示了添加了定时发送邮件的请假流程。

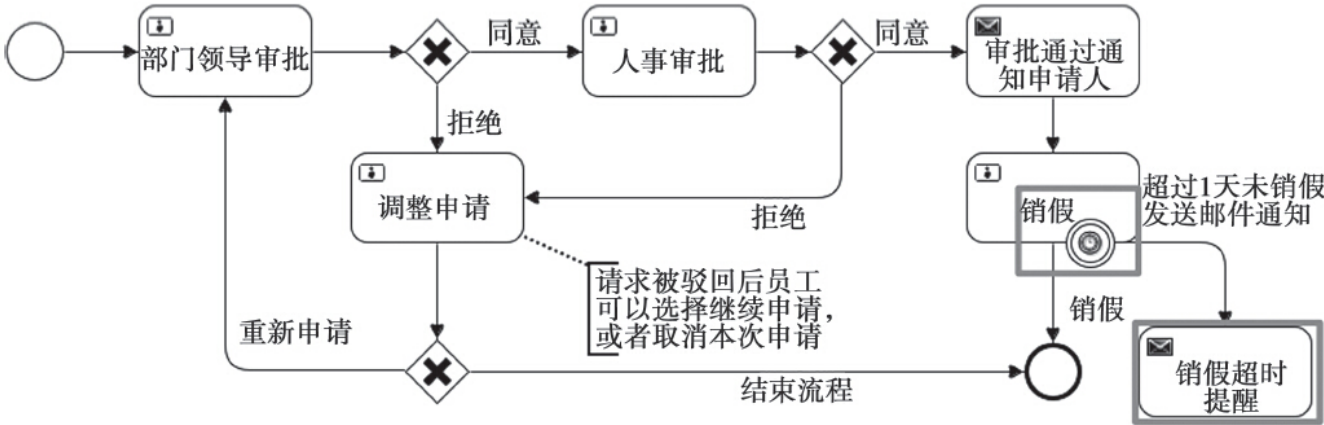


图8-16 添加了定时发送邮件的请假流程

提示 图8-16的流程定义文件路径为chapter8/leave-mail-timeout.bpmn。

如果流程中有定时任务需要开启引擎的Job功能，配置如下：

```
<property name="jobExecutorActivate" value="true"/>
```

代码清单8-8列出了图8-16中添加的定时器边界事件及其输出流“销假超时提醒”的XML代码片段。

代码清单8-8 图8-16中添加的边界事件及销假超时提醒邮件任务的XML代码

```
<boundaryEvent id="boundarytimer1" name="Timer" attachedToRef="reportBack"
  cancelActivity="true" #1
  <timerEventDefinition // ${reportBackTimeout} 表达式用来获取发送邮件的任务
    <timeDate>${reportBackTimeout}</timeDate>
  </timerEventDefinition>
</boundaryEvent>
<serviceTask id="reportBackTimeoutMail" name="销假超时提醒" activiti:type="mail">
  <extensionElements>
    <activiti:field name="to">
      <activiti:expression>${to}</activiti:expression>
    </activiti:field>
    <activiti:field name="from">
      <activiti:string>noreply@localhost</activiti:string>
    </activiti:field>
    <activiti:field name="subject">
      <activiti:string>销假提醒</activiti:string>
    </activiti:field>
    <activiti:field name="charset">
      <activiti:string>UTF-8</activiti:string>
    </activiti:field>
    <activiti:field name="html">
      <activiti:expression>&lt;html&gt;
&lt;body&gt; // &lt;表示符号, &gt;表示符号, Activiti Designer自动做了字符转义
你好 ${name},&lt;br&gt;&lt;br&gt;
&lt;p&gt;您有一个请假申请需要销假, 请及时处理。&lt;/p&gt;
&lt;br&gt;
&lt;div&gt;开始时间: ${startDate}&lt;/div&gt;
&lt;div&gt;结束时间: ${endDate}&lt;/div&gt;
&lt;div&gt;请假原因: ${reason}&lt;/div&gt;
&lt;/body&gt;
      </activiti:expression>&lt;/html&gt;
    </activiti:field>
  </extensionElements>
</serviceTask>
```

```
</body>
</html></activiti:expression>
</activiti:field>
</extensionElements>
</serviceTask>
<sequenceFlow id="flow14" name="超过1天未销假发送邮件通知" sourceRef="boundarytimer1" targetRef="reportBackTimeoutMail"></sequenceFlow>
```

代码清单8-8和代码清单8-6的配置大致一样，#1处需要着重说明的就是定时器边界事件的cancelActivity属性，该属性值可以设置为true或false。

假设当前请假的结束时间为2013-01-01，那么定时发送提醒邮件的日期应该为2013-01-02，这种假设下面两种情况。

- 设置为true时：如果用户在2013-01-02日之前办理了“销假”任务，则会取消“销假”节点的定时器边界事件，也就是从Job列表中删除定时器边界事件。
- 设置为false时：不管用户在2013-01-02日之前还是之后办理了“销假”任务，都会执行边界事件，也就是说“销假”任务完成后不会取消相关的Job，所以引擎会在2013-01-0200:00:00准时发送邮件。

cancelActivity的值要根据业务的实际需求进行设置，设定为false有可能会任务已经办理完了但用户还是会收到邮件。

为了设置定时发送邮件的时间，需要在代码清单8-7的基础上添加一段代码，根据请假结束时间+1天计算得出一个新的时间，并将其设置到变量reportBackTimeout中。

```
// 超时提醒时间设置，请假结束时间+1天
Date endDate = (Date) execution.getVariable("endDate");
Calendar ca = Calendar.getInstance();
ca.setTime(endDate);
ca.add(Calendar.DAY_OF_MONTH, 1);
execution.setVariableLocal("reportBackTimeout", ca.getTime());
```

图8-17展示了一封销假提醒邮件，从中可以看出请假的结束时间为2013-01-01，邮件是在2013-01-0215:19:00发送的（手动调整了系统时间来模拟到期以触发定时任务的执行）。



图8-17 销假提醒邮件

8.3 本章小结

本章的内容只有一个——邮件任务，它不是BPMN 2.0规范中定义的，而是Activiti引擎在ServiceTask上扩展的一个特殊任务类型，但是由于邮件在企业应用中使用比例非常大，因此为了更好地将邮件任务和Activiti结合起来专门用一个章的篇幅进行讲解。

本章开头先使用James模拟企业的邮件系统，使用最简单的方式触发引擎的邮件任务；另外针对企业邮箱加密的情况也使用Gmail为例做了讲解。

8.2节是本章的核心内容，分别讲解了即时发送邮件和定时发送邮件，带领读者学习了如何使用邮件模板+流程变量的方式配置邮件内容，以及动态获取邮件接收人等信息。除了介绍邮件任务的基础使用之外，还针对企业中常用的定时邮件进行了演示，定时器边界事件自动触发邮件的发送，并对定时器边界事件的cancelActivity属性的取值做了详细的讲解，以灵活应对不同需求。

第9章 多实例

多实例（4.3.12节）特性在企业应用中被广泛应用，因为它允许多个用户协作完成一个任务（不仅仅是任务，多实例支持很多种类型的活动），或者批量处理某项任务等。

配置了多实例特性的活动在流程运行时依次（顺序方式）或单次批量（并行方式）创建活动实例，实例的数量由不同的参数决定（参考表4-10）。

会签，顾名思义，就是主办人（单位）联合其他有关人员（部门）联合办理的过程，例如主办单位下发公文（简称发文），一般需要多个部门共同审核通过之后才允许对外发布。再举一个例子——员工离职，需要各个部门签字、盖章确认后方可离职，这一过程也称为“会签”。

本章将详细介绍多实例的适用场景（会签），以及如何在Activiti配置多实例，还介绍多实例的顺序执行与并行执行方式的区别。

9.1 非用户任务

创建多实例，根据任务类型的不同有着不同的方式。非用户任务（例如Java Service任务）可以直接指定任务实例的数量（引擎内部会循环创建任务实例），也就是多实例任务执行的次数；用户任务（UserTask）如果也像非用户任务一样指定任务的数量就显得没有意义了，因为用户任务没有任务的办理人（9.2节会介绍用户任务创建多实例的方法）。

因此，非用户任务创建多实例任务，可以通过配置多实例任务的nrOfInstances参数来实现，当然该参数可以配置为固定值（一般不会这么设置），也可以通过变量方式动态配置。

代码清单9-1 Java Service任务多实例的XML代码

```
<serviceTask id="servicetask1" name="Service Task"
activiti:expression="${counter + 1}" activiti:resultVariableName="counter">
  <multiInstanceLoopCharacteristics isSequential="false"> // 多个实例并行执行
    <loopCardinality>${loop}</loopCardinality>
  </multiInstanceLoopCharacteristics>
</serviceTask>
```

在代码清单9-1中，对一个Java Service任务添加了多实例配置，并且通过设置属性isSequential为false决定多个实例并行执行；通过变量loop决定实例的数量；每次执行完一个实例，计数器加1再写回到counter中。针对此例的测试用例如代码清单9-2所示。

代码清单9-2 代码清单9-1的测试用例

```
@Test
@Deployment(resources = { "diagrams/chapter9/testMultiInstanceFixedNumbers.bpmn" })
public void testParallel() throws Exception {
    Map<String, Object> variables = new HashMap<String, Object>();
    long loop = 3; // 共创建3个实例
    variables.put("loop", loop);
    variables.put("counter", 0); // 计数器从0开始
    ProcessInstance processInstance = runtimeService.startProcessInstanceByKey("testMultiInstanceFixedNumbers", variables);
    Object variable = runtimeService.getVariable(processInstance.getId(), "counter");
    assertEquals(loop, variable); // 计数器的值与loop变量相等，说明执行了3次
}
```

代码清单9-2对Java Service任务执行了三次验证，由于Java Service任务是自动执行的，因此当流程启动之后三个实例并行执行（相当于开启了三个子线程），最后变量counter的值与loop值相等。

将本例中Java Service任务的多实例属性isSequential设置为true或者false，其结果总是一致。这是因为Java Service任务在创建之后立即执行了任务完成动作，当设置顺序执行（isSequential设置为true）时，执行的过程是一个接一个地执行，所以结果counter的值还是3。

对于4.3.12节列出的几个针对多实例的内置变量，可以直接应用于表达式中，当然也可以在任务监听类中获取内置变量以便协助处理业务逻辑。内置变量有：nrOfInstances、nrOfActiveInstances、nrOfCompletedInstances、loopCounter。

 **提示** 邮件任务的多实例和Java Service任务类似，如果需要向一批用户发送邮件，那么可以把邮件任务的to属性通过表达式方式获取，可以根据当前执行的实例对象获取不同的邮件地址列表，然后依次或执行发送邮件的任务。

9.2 用户任务多实例

和Java Service一样，如果用户任务也通过设置loopCardinality元素的值来决定实例数量，就显得毫无意义，因为缺少了用户任务的主要特性——人，所以用户任务实例数量就需要由参与的人数决定。

对于本章开头提到的会签，一般由主办人发起，从用户组织树中选择几个参与人一同办理任务，如果用户任务设置了按照并行的方式处理，那么将一次创建多个（和选择的人数数量相等）实例，并按照选择参与人的顺序依次把任务分配（assignee）给参与人。如果按照顺序的方式执行任务，首先会创建一个任务实例分配给第一个参与人（假设序号从1开始），只有在第一个参与人办理完任务之后才会创建第二个任务实例、第三个任务实例……，并依次按照参与人的顺序设定任务的办理人。

对于不同业务的联合审批有着不同的处理方式，下面分别介绍两种联合审批在流程中如何设定及协作。

9.2.1 顺序方式办理

顺序办理也可以称为排队办理，只有一个人办理完后面的人才可以继续办理。比如使用纸质方式申请请假，需要填写请假条，然后拿着这张假条找领导、人事或者总经理审批，但是每次只能找一个人签字确认，顺序方式在流程引擎中也是如此。

代码清单9-3列出了一个包含顺序多实例用户任务的流程定义。

代码清单9-3 用户任务多实例——顺序执行流程定义的XML代码

```
<process id="testMultiInstanceForUserTask" name="testMultiInstanceForUserTask" isExecutable="true">
  <startEvent id="startevent1" name="Start"></startEvent>
  <userTask id="usertask1" name="User Task" activiti:assignee="${user}"> #1
    <multiInstanceLoopCharacteristics isSequential="true" // 顺序执行 #2
      activiti:collection="${users}" // 用户集合，直接写作"users"也可以 #3
      activiti:elementVariable="user" // 遍历users变量的单个对象变量名称 #4
    ></multiInstanceLoopCharacteristics>
  </userTask>
</process>
```

代码清单9-3在#2处设置多实例元素的“isSequential”属性为true，表示多个实例按照#3处activiti:collection指定的集合变量名称的顺序执行每个实例，在遍历users集合时把单个值保存在#4处属性“activiti:elementVariable”的“user”中，如此便可在#1处使用该变量读取并设置任务的办理人属性。代码清单9-4通过测试代码验证了执行过程。

代码清单9-4 代码清单9-3的测试用例

```
@Test
@Deployment(resources = { "diagrams/chapter9/testMultiInstanceForUserTask.users.sequential.bpmn" })
```

```
public void testForUserCreateByUsersSequential() throws Exception {
    Map<String, Object> variables = new HashMap<String, Object>();
    List<String> users = Arrays.asList("user1", "user2", "user3"); // 设置会签的3个用户 #1
    variables.put("users", users);
    runtimeService.startProcessInstanceByKey("testMultiInstanceForUserTask", variables);
    for (String userId : users) { // 遍历3次
        Task task = taskService.createTaskQuery().taskAssignee(userId).singleResult();
        taskService.complete(task.getId());
    }
}
```

代码清单9-4在流程启动时设置了变量“users”，在for循环中每次读取一个任务（当前也只有一个任务）并完成任务的办理，如此遍历完users集合也结束了流程。

图9-1清晰地描述了这一过程。

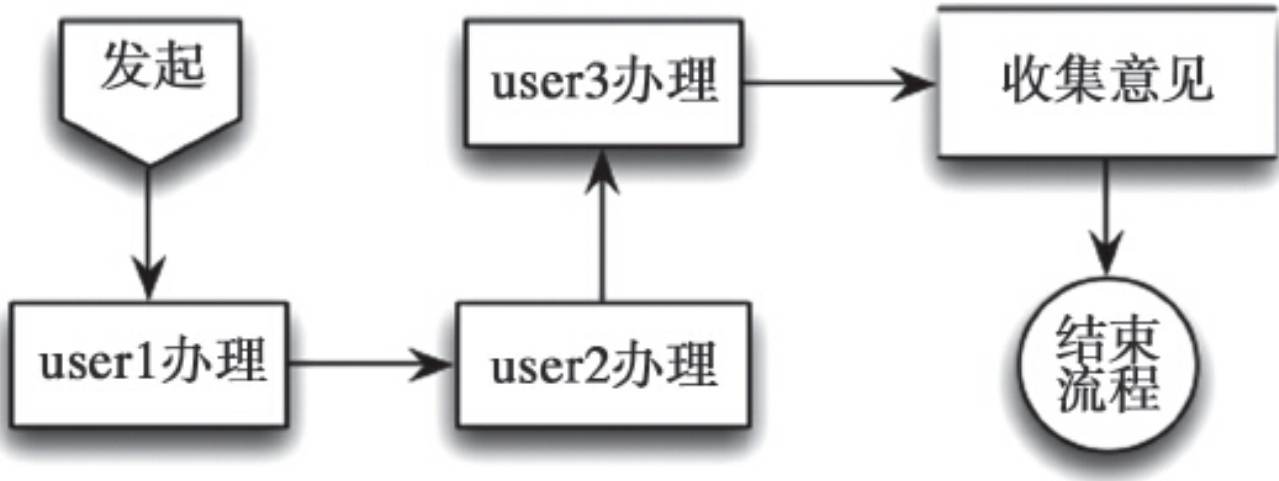


图9-1 多实例顺序执行的过程

9.2.2 并行方式办理

相对于顺序办理方式来说，并行方式办理的效率要高得多，好比多线程下载工具。图9-2清晰地描述了这一过程。

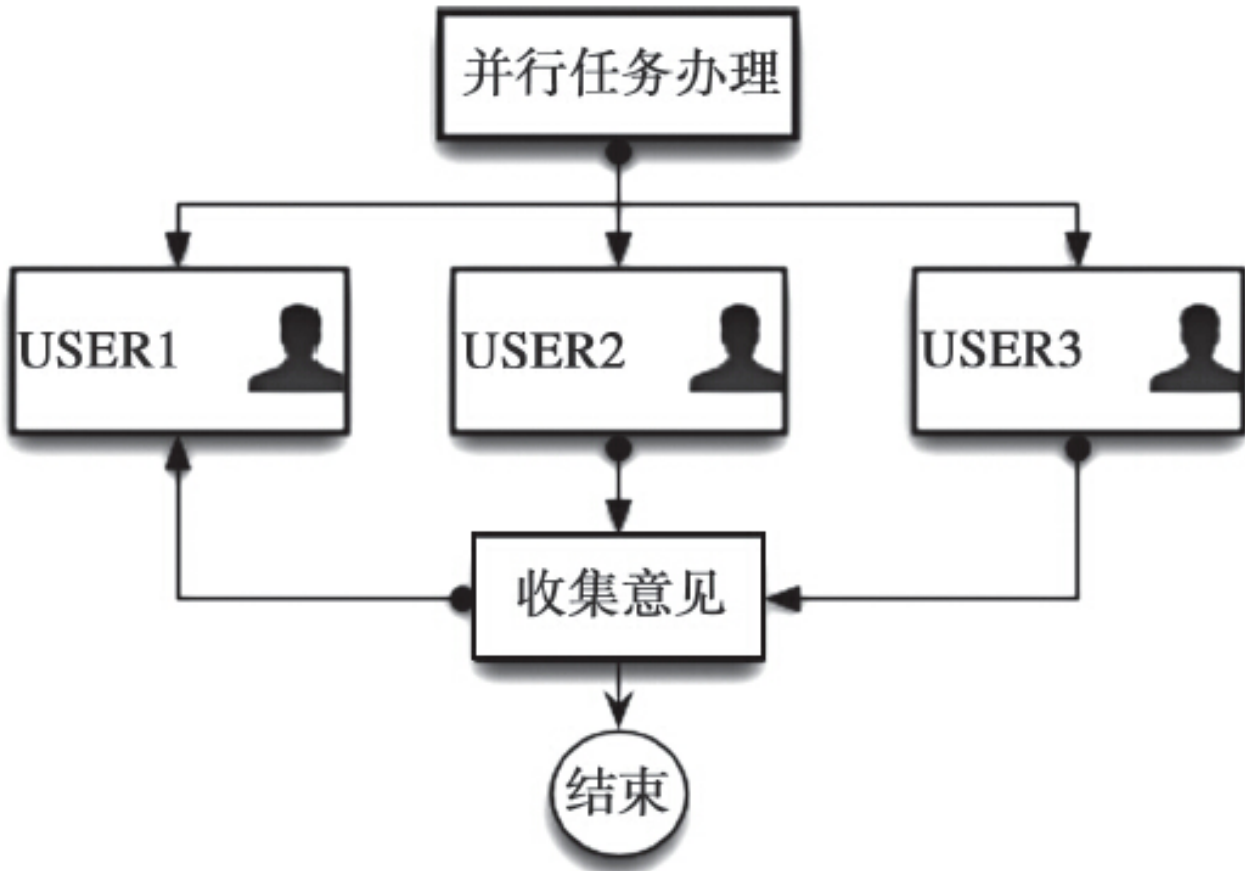


图9-2 多实例并行执行的过程

代码清单9-5列出了一个包含并行多实例用户任务的流程定义。

代码清单9-5 用户任务多实例——并行执行流程定义的XML代码

```
<process id="testMultiInstanceForUserTask" name="testMultiInstanceForUserTask" isExecutable="true">
  <startEvent id="startevent1" name="Start"></startEvent>
  <userTask id="usertask1" name="User Task" activiti:assignee="{user}"> #1
```

```
<multiInstanceLoopCharacteristics isSequential="false" // 并行执行 #2
    activiti:collection="{users}" // 用户集合，直接写作“users”也可以 #3
    activiti:elementVariable="user" // 遍历users变量的单个对象变量名称 #4
></multiInstanceLoopCharacteristics>
</userTask>
</process>
```

代码清单9-5和代码清单9-3除了#2处属性“isSequential”的值不同之外，其他都一样，区别在于并行方式在启动流程之后会一次产生3条任务并分别设置任务办理人为user1、user2、user3。代码清单9-6的单元测试可以验证代码清单9-5的执行过程。

代码清单9-6 代码清单9-5的测试用例

```
@Test
@Deployment(resources = { "diagrams/chapter9/testMultiInstanceForUserTask.users.nosequential.bpmn" })
public void testForUserCreateByUsersNoSequential() throws Exception {
    Map<String, Object> variables = new HashMap<String, Object>();
    List<String> users = Arrays.asList("user1", "user2", "user3");
    variables.put("users", users);
    runtimeService.startProcessInstanceByKey("testMultiInstanceForUserTask", variables);
    for (String userId : users) { // 遍历验证每一个用户都有一个任务
        assertEquals(1, taskService.createTaskQuery().taskAssignee(userId).count());
    }
}
```

验证的过程主要在代码清单9-6的for循环体中，每次验证一个用户有一个待办任务，即表示在#1处启动流程后系统已经产生了3条待办任务，和图9-2中的3个用户相吻合。

9.2.3 设置结束条件

只有在把所有引擎创建的任务都办理完成后9.2.1和9.2.2介绍的两种方式才会继续执行后续输出流，如果需要根据不同的比例提前结束会签，该如何做呢？

现实中有投票的例子，比如投票通过率为60%即认为通过，可以提前结束投票。在BPMN 2.0规范中允许设置多实例的结束条件，当条件表达式值为true时结束该任务的执行。

代码清单9-7展示了一个模拟投票的流程，投票比例大于等于60%即认为通过。

代码清单9-7 设置了结束条件的多实例

```
<process id="testMultiInstanceForUserTask" name="testMultiInstanceForUserTask" isExecutable="true">
    <startEvent id="startevent1" name="Start"></startEvent>
    <userTask id="usertask1" name="User Task" activiti:assignee="{user}">
        <multiInstanceLoopCharacteristics isSequential="true" activiti:collection="{users}"
            activiti:elementVariable="user">
            <completionCondition> // 设置结束此任务的条件
                ${nrOfCompletedInstances / nrOfInstances >= rate} #1
            </completionCondition>
        </multiInstanceLoopCharacteristics>
    </userTask>
</process>
```

相对于前面的两个例子，代码清单9-7添加了completionCondition元素用于设置一个表达式，通过引擎内置变量（只对多实例任务有效）的值获取任务完成比例和预期的比率（rate变量）相比较的结果，如果值为true，则结束该任务的执行。对应的测试代码见代码清单9-8所示。

代码清单9-8 代码清单9-7的测试用例

```
@Test
@Deployment(resources = { "diagrams/chapter9/testMultiInstanceForUserTask.users.sequential.with.complete.conditon.bpmn" })
public void testForUserCreateByUsersSequentialWithCompleteCondition() throws Exception {
    Map<String, Object> variables = new HashMap<String, Object>();
    List<String> users = Arrays.asList("user1", "user2", "user3");
    variables.put("users", users);
    variables.put("rate", 0.6d); // 设置60%的比率作为完成条件 #1
    runtimeService.startProcessInstanceByKey("testMultiInstanceForUserTask", variables);
    // 第1个用户完成任务
    Task task = taskService.createTaskQuery().taskAssignee("user1").singleResult();
    taskService.complete(task.getId()); // #2
    // 第2个用户完成任务
    task = taskService.createTaskQuery().taskAssignee("user2").singleResult();
    taskService.complete(task.getId()); // #3
    // 此时该任务已经结束，因为该流程只有一个任务，所以流程结束
    long count = historyService.createHistoricProcessInstanceQuery().finished().count();
    assertEquals(1, count);
}
```

从代码清单9-8中可以看出，任务“usertask1”分配了三个用户，但是只在#2和#3处执行了任务动作，在#4处读取已经结束的流程结果为1，这样的结果是在#1处设置的rate变量以及代码清单9-7中#1处的表达式在每次任务完成（complete）时被调用并计算所致。

9.3 应用实例——请假会签

通过前面几个单元测试相信大家基本了解了多实例的配置与工作方式，本节将以请假会签为例展示多实例在Web应用中的应用，包括如何在流程启动时选择参与者（对于OA系统中的发文流程来说相当于主人）。

9.3.1 流程定义

本例的流程定义是在第6章动态表单的基础上合并了“部门领导”和“人事审批”节点为“部门/人事联合会签”，新的流程图如图9-3所示。对应的请假会签流程定义的XML代码如代码清单9-9所示。

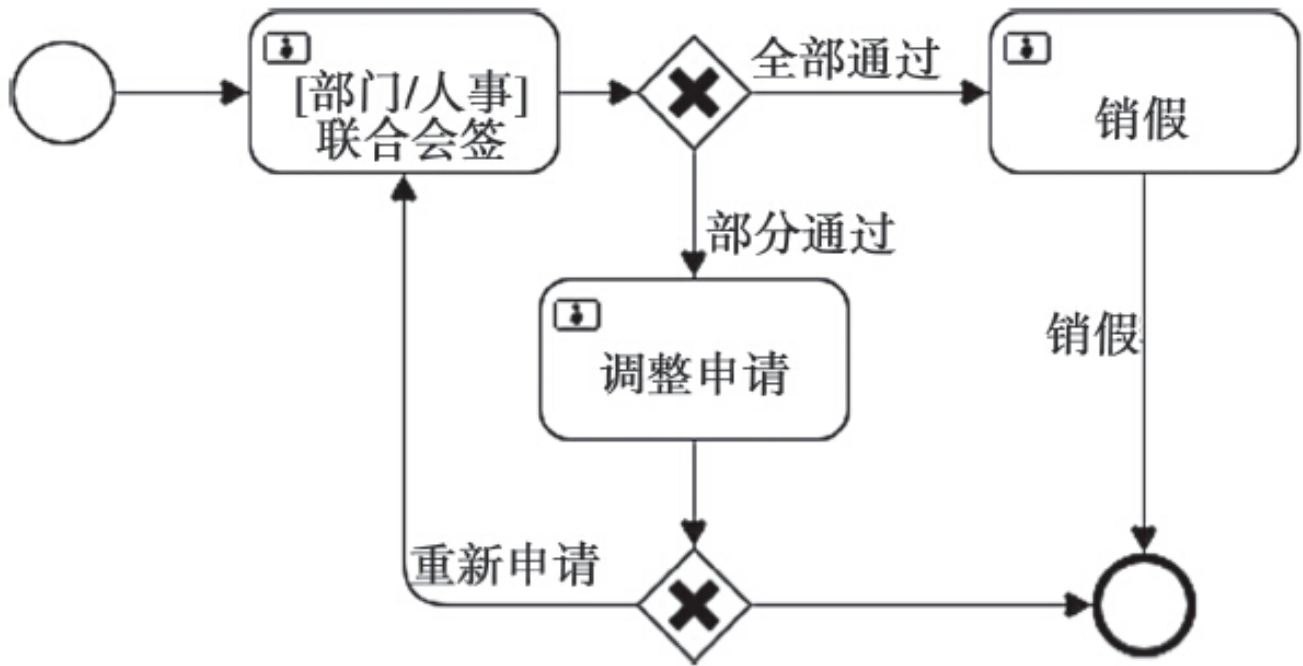


图9-3 请假会签流程图

代码清单9-9 请假会签流程定义的XML代码

```

<process id="leave-countersign" name="请假流程-会签" isExecutable="true">
  <documentation>请假流程演示-会签</documentation>
  <startEvent id="startevent1" name="Start" activiti:initiator="applyUserId">
    <extensionElements>
      <activiti:formProperty id="startDate" name="请假开始日期" type="date" datePattern="yyyy-MM-dd" required="true"></activiti:formProperty>
      <activiti:formProperty id="endDate" name="请假结束日期" type="date" datePattern="yyyy-MM-dd" required="true"></activiti:formProperty>
      <activiti:formProperty id="reason" name="请假原因" type="string" required="true"></activiti:formProperty>
      // 审批参与人字段的type为自定义类型"users", 实现类见代码清单9-10
      <activiti:formProperty id="users" name="审批参与人" type="users" /> #1
      <activiti:formProperty id="validScript" type="javascript" default="alert('表单已经加载完毕');"></activiti:formProperty>
    </extensionElements>
  </startEvent>
  <userTask id="countersign" name="[部门/人事]联合会签" activiti:assignee="${user}">
    <extensionElements>
      // 省略了请假的时间、请假原因字段配置
      // 通过多实例的user变量设置任务办理人
      <activiti:formProperty id="approved" name="审批意见" type="enum">
        <activiti:value id="true" name="同意"></activiti:value> // 单次审批的结果
        <activiti:value id="false" name="拒绝"></activiti:value> // 保存在变量approved中
      </activiti:formProperty>
      <activiti:taskListener event="complete" // 单个处理人办理完任务之后
        delegateExpression="${leaveCounterSignCompleteListener}" />
    </extensionElements>
    <multiInstanceLoopCharacteristics isSequential="false" // 并行执行多实例
      activiti:collection="users" activiti:elementVariable="user" /> // 设置用户任务的办理人
  </userTask>
  <userTask id="reportBack" name="销假" activiti:assignee="${applyUserId}">
    <extensionElements>
      // 省略了请假的时间、请假原因、销假时间字段配置
    </extensionElements>
  </userTask>
  <endEvent id="endevent1" name="End"></endEvent>
  <sequenceFlow id="flow2" sourceRef="startevent1" targetRef="countersign">
    <extensionElements>
      <activiti:executionListener event="take" // 初始化变量approvedCounter为0
        expression="${execution.setVariable('approvedCounter', 0)}" />
    </extensionElements>
  </sequenceFlow>
  <sequenceFlow id="flow8" name="销假" sourceRef="reportBack" targetRef="endevent1">
    <extensionElements>
      <activiti:executionListener event="take"
        expression="${execution.setVariable('result', 'ok')}" />
    </extensionElements>
  </sequenceFlow>
  <sequenceFlow id="flow13" name="全部通过" sourceRef="exclusivegateway1" targetRef="reportBack">
    <conditionExpression xsi:type="tFormalExpression"> // 全部同意申请, 即参与会签的
      <![CDATA[${approvedCounter == users.size()}]]> // 人数和同意申请的数量相等,
    </conditionExpression> // 即通过率为100%
  </sequenceFlow>
  <exclusiveGateway id="exclusivegateway1" name="Exclusive Gateway" />
  <sequenceFlow id="flow14" sourceRef="countersign" targetRef="exclusivegateway1" />
  <userTask id="modifyApply" name="调整申请" activiti:assignee="${applyUserId}"></userTask>
  <sequenceFlow id="flow15" name="部分通过" sourceRef="exclusivegateway1" targetRef="modifyApply">
    <conditionExpression xsi:type="tFormalExpression"> // 部分通过 (包含全部未通过)
      // 审批通过的计数器值小于参与人的数量
      <![CDATA[${approvedCounter < users.size()}]]>
    </conditionExpression>
  </sequenceFlow>
  <exclusiveGateway id="exclusivegateway2" name="Exclusive Gateway" />
  <sequenceFlow id="flow16" sourceRef="modifyApply" targetRef="exclusivegateway2" />
  <sequenceFlow id="flow17" sourceRef="exclusivegateway2" targetRef="endevent1">
    <conditionExpression xsi:type="tFormalExpression">
      <![CDATA[${reApply == 'false'}]]></conditionExpression>
  </sequenceFlow>
  <sequenceFlow id="flow18" name="重新申请" sourceRef="exclusivegateway2" targetRef="countersign">
    <extensionElements>
      <activiti:executionListener event="take" // 重新申请时把变量approvedCounter置为0
        expression="${execution.setVariable('approvedCounter', 0)}" />
    </extensionElements>
    <conditionExpression xsi:type="tFormalExpression">
      <![CDATA[${reApply == 'true'}]]></conditionExpression>
  </sequenceFlow>
</process>

```

代码清单9-9看起来比较长, 读者可以把重点放在加粗和有中文注释的地方, 这样比较容易理解。结合图9-3和代码清单9-9可以得出如下描述文字:

- 启动流程时把计数器approvedCounter设置为0并选择参与人。
- 在每个参与人办理完任务后由任务的complete事件类型的监听器处理结果并保存, 当同意时把变量approvedCounter的值加1。

· 在所有的办理人都处理完成后执行排他路由（id为exclusivegateway1），如果同意计数器（approvedCounter）和参与人数量相等，即表示全部通过，否则退回至“调整申请”。

#1处的字段“审批参与人”的类型为“users”，这不是引擎支持的字段类型，而是根据需要自定义的一个表单字段类型（6.2.4节）。代码清单9-10列出了该字段类型的实现代码。

代码清单9-10 请假会签流程中表单字段类型users的实现

```
public class UsersFormType extends AbstractFormType {
    @Override
    public String getName() { return "users"; }
    @Override
    public Object convertFormValueToModelValue(String propertyValue) {
        String[] split = StringUtils.split(propertyValue, ",");
        return Arrays.asList(split); // 把字符串型的值转换为List集合对象
    }
    @Override
    public String convertModelValueToFormValue(Object modelValue) {
        return ObjectUtils.toString(modelValue); // 把集合对象转换为字符
    }
}
```

在6.2.4节中，表单自定义类型需要向引擎注册才能使用，下面的配置就是把代码清单9-10的表单字段类添加到引擎的自定义表单字段类型集合中。

```
<property name="customFormTypes">
    <list>
        <bean class="me.kafeitu.activiti.chapter6
            .form.JavascriptFormType" />
        <bean class="me.kafeitu.activiti.chapter9
            .form.UsersFormType" />
    </list>
</property>
```

9.3.2 任务办理

由于在Spring配置文件中配置了自动部署，因此在服务启动完成之后会在流程定义列表中存在一条名称为“请假会签”的流程定义记录（如图9-4所示），自动部署的配置如下：

```
<property name="deploymentResources">
    <list>
        <value>classpath*:chapter8/leave-mail*.zip</value>
        <value>classpath*:chapter9/leave-countersign.zip</value>
    </list>
</property>
```

流程定义ID	部署ID	流程定义名称	流程定义KEY	流程描述	版本号	XML资源名称	图片资源名称	操作
leave-countersign:1:10	1	请假流程-会签	leave-countersign	请假流程演示-会签	1	leave-countersign.bpmn	leave-countersign.png	删除 启动

图9-4 已部署的请假会签流程

用账号“henry”登录，然后单击图9-4中的“启动”按钮即打开启动流程的界面，在此界面中需要填写任务表单，如图9-5所示。

启动流程 — [请假流程-会签]，版本号：1

请假开始日期:

2013-01-15

请假结束日期:

2013-01-18

请假原因:

回老家

审批参与人:

◀ 返回列表

▶ 启动流程

图9-5 启动请假会签流程

图9-5中的“审批参与人”字段显示为灰色，表示该字段只读。在对代码清单9-9中的流程定义进行解读时对该字段进行过说明，这是一个自定义类型的字段，用来保存该流程的参与人，由代码清单9-10的实现类进行类型转换。

为了模拟企业应用中从组织树中选择人员的功能特意设计了一个功能（利用Bootstrap的Modal展示），可以动态地读取系统中的人员并按照角色分组显示，用户可以多选，单击对话框的“确定”按钮后把选择的用户ID设置到“审批参与人”输入框中。单击“审批参与人”输入框后会弹出如图9-6的对话框，可以按住Ctrl键多选。

选择人员



部门经理

Andy Zhao (andy)

Kermit Miao (kermit)

管理员

Henry Yan (henry)

总经理

Bill Zheng (bill)

人事经理

Jenny Luo (jenny)

关闭

确定

图9-6 从对话框中选择会签参与人

从图9-6中可以看出，选择了两个用户，分别是部门经理kermit和人事经理jenny，单击“确定”按钮后“审批参与人”输入框的值为“kermit, jenny”，如图9-7所示。

审批参与人:

图9-7 选择了审批参与人

在完成表单填写后单击图9-5的“启动流程”按钮触发流程启动动作，然后分别以kermit和jenny身份登录系统可以在“任务列表”模块看到一个待办任务。从图9-8可以清楚地看出两个任务的“流程实例ID”相同。

任务ID	任务名称	流程实例ID	流程定义ID	任务创建时间	办理人	操作
125	[部门/人事]联合会签	101	leave-countersign:1:10	Tue Jan 15 01:09:53 CST 2013	kermit	

任务ID	任务名称	流程实例ID	流程定义ID	任务创建时间	办理人	操作
129	[部门/人事]联合会签	101	leave-countersign:1:10	Tue Jan 15 01:09:53 CST 2013	kermit	

图9-8 kermit和jenny分别有一个联合会签待办任务

接下来分别办理待办任务，在“审批意见”处选择“同意”，表示该申请全部通过。这样在kermit和jenny都办理完成任务后流程流转至“销假”节点。任务办理表单如图9-9所示。

任务办理—[[部门/人事]联合会签]，流程定义ID：
[leave-countersign:1:10]

请假开始日期:

请假结束日期:

请假原因:

审批意见:



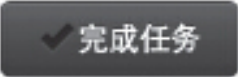


图9-9 会签任务办理表单

最后用户“henry”登录，在“任务列表”可以看到一条名称为“销假”的待办任务，如图9-10所示。

任务ID	任务名称	流程实例ID	流程定义ID	任务创建时间	办理人	操作
16	销假	101	leave-countersign:1:10	Tue Jan 15 01:18:19 CST 2013	henry	

图9-10 联合会签结束后流程流转至销假节点

9.4 审批意见

9.3.2节的执行过程都是申请通过的情况，如果审批不通过该如何处理呢？肯定需要驳回的理由，这样申请人在收到被驳回的请求时可以清楚地知道为什么申请被驳回了。

审批意见是工作流引擎中一个不可缺少的模块。流程变量可以保存流程处理过程中的中间状态（一般用于条件判断），审批意见可以保存每个任务（一般指用户任务）办理时产生的意见。例如请假不予批准时领导通过填写审批意见，在“调整申请”节点读取并展示给申请人。

本节内容将围绕审批意见进行介绍，利用请假流程的审批讲解如何使用Activiti的审批意见（Activiti称之为Comment，也可以称之为批注）功能。

Activiti在TaskService接口中定义了3个和意见有关的方法：

- addComment(String taskId, String processInstanceId, String message)
- getProcessInstanceComments(String processInstanceId)
- getTaskAttachments(String taskId)

第一个方法用于创建意见，第二个和第三个方法分别根据流程实例ID和任务ID读取相关意见。从这3个方法可以看出意见总是和任务有关，而人可以参与的任务只有用户任务，所以意见依附于用户任务。

图9-11在原来的任务办理界面添加了保存意见、读取意见列表的功能。



图9-11 添加了保存意见和读取意见功能的任务办理界面

代码清单9-11中的两个方法分别用于保存意见和读取意见，并且从用户体验角度考虑使用Ajax方式发送请求。

代码清单9-11 保存意见的Controller方法

```
@Controller
@RequestMapping(value = "/chapter9/comment")
public class CommentController {
    // 保存意见
    @RequestMapping(value = "save", method = RequestMethod.POST)
    @ResponseBody // 保存后返回true, 为Ajax请求服务
    public Boolean addComment(@RequestParam("taskId") String taskId,
        @RequestParam("processInstanceId") String processInstanceId,
        @RequestParam("message") String message, HttpSession session) {
        // 设置当前人, 引擎会把当前人作为意见的所属人
        identityService.setAuthenticatedUserId(UserUtil.getUserFromSession(session).getId());
        // 调用保存意见的方法
        taskService.addComment(taskId, processInstanceId, message);
        return true;
    }
    @RequestMapping(value = "list/{processInstanceId}")
    @ResponseBody
    public Map<String, Object> list(@PathVariable("processInstanceId") String processInstanceId) {
        Map<String, Object> result = new HashMap<String, Object>();
        List<Comment> taskComments = taskService // 根据流程实例ID读取意见
            .getProcessInstanceComments(processInstanceId);
        List<HistoricTaskInstance> list = historyService.createHistoricTaskInstanceQuery()
            .processInstanceId(processInstanceId).list();
        Map<String, String> taskNames = new HashMap<String, String>();
        for (HistoricTaskInstance historicTaskInstance : list) {
            // 读取历史任务以键值对方式保存任务ID和名称
            taskNames.put(historicTaskInstance.getId(), historicTaskInstance.getName());
        }
        result.put("comments", taskComments);
        result.put("taskNames", taskNames);
    }
}
```

```
    return result;
  }
}
```

文件chapter9/src/main/webapp/WEB-INF/view/chapter6/task-form.jsp包含了对应的Ajax请求的代码。根据流程实例ID读取意见列表接收到的JSON格式数据如图9-12所示。

在会签处理完毕之后任务流转至“调整申请”节点，此时申请人就可以明确为什么申请被驳回，可以根据审批意见做出处理。图9-13展示了调整申请节点的办理页面。



图9-12 读取意见列表接收到的JSON数据



图9-13 调整申请任务办理界面（有意见列表）

9.5 本章小结

本章内容以请假会签为例讲解了Activiti对多实例的支持，先介绍了顺序和并行两种执行方式，用单元测试和实例阐释了两者的不同。对于设置结束条件的多实例任务可以设置完成条件提前结束任务。最后介绍了审批意见在流程中的应用，通过实例展示了如何保存和当前任务相关的意见，以及如何读取意见列表，如此可以清楚地知道参与人在任务办理过程中的不同意见。

第10章 子流程与调用活动

《重构：改善既有代码的设计》一书中提到了LongMethod（过长方法）的重构，有编码经验的程序员都会下意识地去重构自己的代码，让其可维护性更高、可读性更好。当一个方法要执行的任务过于复杂时就应该重构代码了，如果一个方法中涉及几个功能的处理，此时就应该把每个功能抽取出来单独处理。举个例子，一根箭很容易折断，一把箭很难折断，人与人的合作就像一把箭，而程序员写代码应该是一根箭，这样在遇到问题的时候可以清晰定位，也便于其他的阅读代码。

写代码和流程处理有什么关系呢？我们再举一个例子说明它们的关系：使用Activiti作为工作流引擎开发了一个系统，业务流程也不算太复杂，随着时间的推移客户的需求变得越来越复杂，每次用户的需求变更后流程设计也随之变动，如此迭代，时间久了流程设计也越来越复杂，久而久之流程就很难维护。

在代码中，一个方法的功能过多时会拆分为多个方法，对于流程设计也是如此，可以从业务层面抽象考虑，把不同阶段的任务抽取出来作为一个子流程（Subprocess）处理，如此当业务发生变化时只需要把更改聚焦在不同的子流程上即可，主流程负责串联（输出流）多个子流程。

我们在开发时经常会使用很多第三方的插件来完成特定功能，例如利用POI处理Excel，一般不会自己去写处理Excel的代码而是利用现有的通用组件完成。在实际业务中也经常会有这样的需求，对于财务工作中的付款流程，付款的步骤基本固定，有申请、审批、出纳付款等节点，如果一个系统中有多处地方需要付款，那就需要在每个业务流程中设计重复的付款流程（或者作为一个子流程）。这样的设计有明显的弊端——重复且难以维护，对于有代码洁癖的人来说是绝对不能容忍的。

调用活动（Call Activity）正是用来解决业务流程中重复设计的问题的，只要设计好一个通用（或共用）的流程后由其他的流程引用即可，当通用的部分变动时，业务流程无需更改，仅更改通用流程即可。

本章内容以采购办公用品为例讲解如何应用子流程与调用活动，以及两者的区别。

10.1 子流程

先了解一下整个办公用品采购的业务流程，如图10-1所示。和前面章节接触到的流程相比，图10-1的流程图稍微“复杂”了一些，涉及子流程（4.5.1节）、边界事件（4.6.1节）、异常结束事件（4.1.2节）。

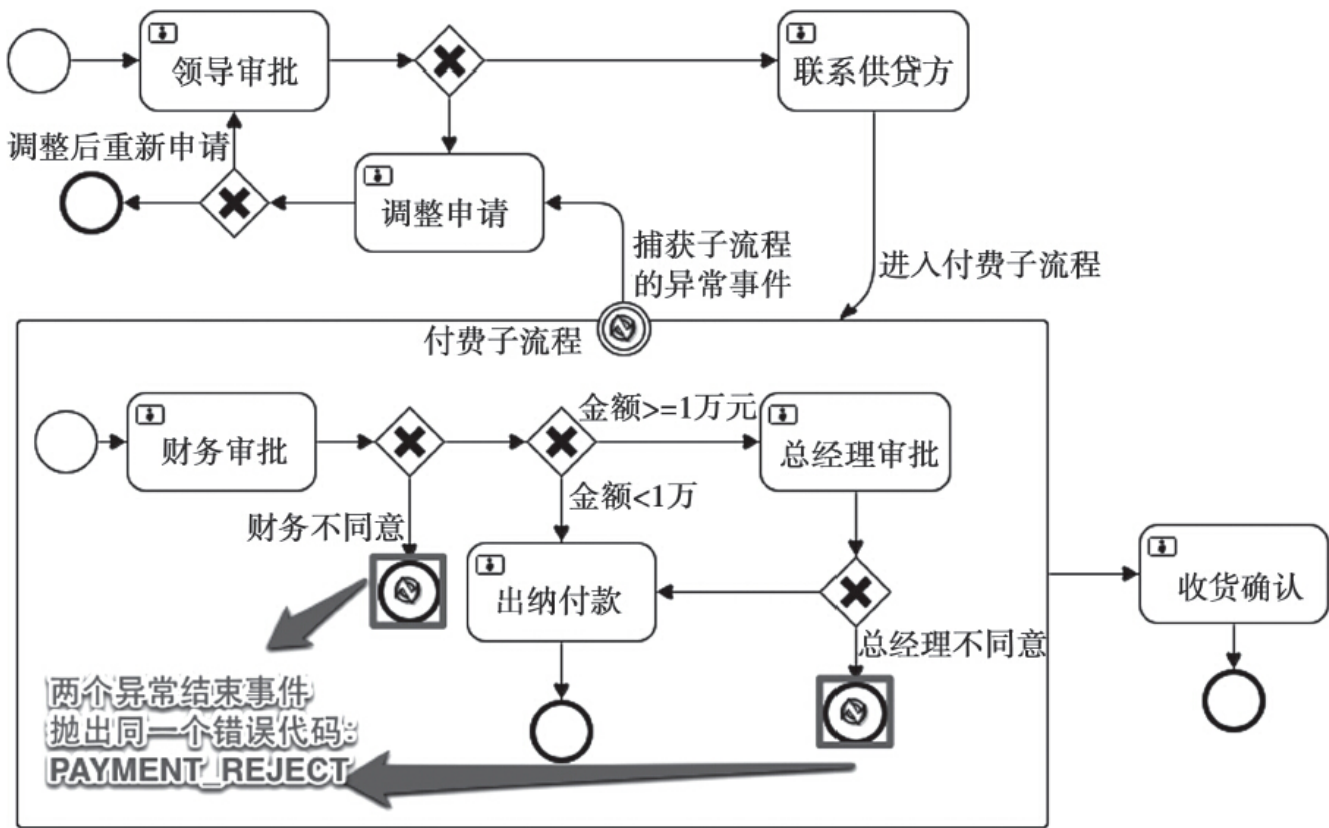


图10-1 办公用品采购流程图

此流程的表单形式以动态表单展示，在启动流程时由申请人填写申请内容，在领导审批通过之后由后勤人员和供货方联系并填写支付货款的账号信息，随即流程进入“付费子流程”，交由财务部门处理，财务部门经过审批后结束“付费子流程”，之后任务流转至“收货确认”节点（将任务分配给该流程的申请人）。

图10-1的方框中即是“付费子流程”，在4.5.1节中做过简单的介绍，该子流程也是一个完整的流程，有启动事件、用户任务、输出流以及结束事件，另外还针对流程的执行异常做了异常抛出处理，见图10-1中箭头标记处的说明。在付费子流程上还附加了一个异常边界事件，用于捕获子流程抛出的流程异常，根据第4章中异常事件与异常边界事件的介绍可以知道：抛出的异常编码如果和捕获方（异常边界事件）定义的错误编码一致，则可以捕获异常编码从而进一步处理，在此流程中抛出与捕获的异常编码均为“PAYMENT_REJECT”，所以当捕获到异常事件之后流程流转至“调整申请”用户任务。

在思考子流程中会有针对金额（图10-2中的“总金额”）的判断，当申请金额大于1万元时要经过“总经理审批”，小于1万元时则不需要。当然，对于排他分支或者其他需要使用表达式的活动来说，都需要以变量的方式提供给流程实例才能计算得出结果。在子流程中判断金额是否大于1万的表达式为“`#{amountMoney}>=10000}`”，其中amountMoney对应图10-2中的“总金额”字段，因为子流程实例可以读取到主流程（也称之为父流程）中的所有变量，所以表达式“`#{amountMoney}>=10000}`”可以正确计算得出结果。

启动流程一 [办公用品采购]，版本号：1

到货期限: 2013-02-27

物品清单: 1. MacBook Pro一台
2. 27寸显示器一台

总金额: 22300

◀ 返回列表

▶ 启动流程

图10-2 启动办公用品采购流程

10.1.1 流程定义

下面分段讲解办公用品采购流程的XML定义，读者可以打开本书配套资源文件的本章目录下的[purchase.bpmn](#)文件查看完整的流程定义。

图10-2所示的流程启动界面的XML代码如下：

```
<startEvent id="startevent1" name="startevent1"
  activiti:initiator="applyUserId">
  <extensionElements>
    <activiti:formProperty id="dueDate" name="到货期限"
      type="date" datePattern="yyyy-MM-dd"
      required="true"/>
    <activiti:formProperty id="listing" name="物品清单"
      type="bigtext" required="true"/>
    <activiti:formProperty id="amountMoney" name="总金额"
      type="double" required="true"/>
  </extensionElements>
</startEvent>
```

其中“物品清单”字段与“总金额”字段的type为自定义。为了满足本例的需求扩展了“bigtext”和“double”两个自定义表单字段类型，见代码清单10-1所示。

代码清单10-1 bigtext与double自定义表单字段类型

```
// 大文本，即html的textarea，继承自引擎默认支持的string类型的实现
public class BigtextFormType extends StringFormType {
    public String getName() {
        return "bigtext";
    }
}
// java.lang.Double表单字段类型
public class DoubleFormType extends AbstractFormType {
    public String getName() {
        return "double";
    }
    public Object convertFormValueToModelValue(String propertyValue) {
        return new Double(propertyValue);
    }
    public String convertModelValueToFormValue(Object modelValue) {
        return ObjectUtils.toString(modelValue);
    }
}
```

最后要把自定义的表单字段类注册到引擎中，下面列出了注册表单字段类的代码，完整的配置参见applicationContext.xml文件。

```
<bean id="processEngineConfiguration" class="org.activiti.spring.SpringProcessEngineConfiguration">
  <property name="customFormTypes">
    <list>
      <bean class="http://www.hzcourse.com/resource/readBook?path=/openresources/teach_ebook/uncompressed/15030/OEBPS/Text/...form.JavascriptFormType" />
      <bean class="http://www.hzcourse.com/resource/readBook?path=/openresources/teach_ebook/uncompressed/15030/OEBPS/Text/...form.UsersFormType" />
      <bean class="http://www.hzcourse.com/resource/readBook?path=/openresources/teach_ebook/uncompressed/15030/OEBPS/Text/... form.DoubleFormType" />
      <bean class="http://www.hzcourse.com/resource/readBook?path=/openresources/teach_ebook/uncompressed/15030/OEBPS/Text/...form.BigtextFormType" />
    </list>
  </property>
</bean>
```

在用户任务“联系供货方”完成后输出流指向“付费子流程”。从下面的代码中可以看出“flow22”的“targetRef”属性为“subprocessPay”，并且在这个顺序流上添加了一个监听器，用来设置变量“usage”的值。


```
<sequenceFlow id="flow22" name="进入付费子流程"
    sourceRef="contactSupplier" targetRef="subprocessPay">
    <extensionElements>
        <activiti:executionListener event="take"
            expression="${execution.setVariable('usage', listing)}"/>
    </extensionElements>
</sequenceFlow>
```

子流程需要使用标签`subProcess`包裹，其内部的活动集合也是一个完整的流程。付费子流程的部分XML代码如下：

```
<subProcess id="subprocessPay" name="付费子流程">
    <startEvent id="startevent2" name="Start"></startEvent>
    <userTask id="generalManagerAudit" name="总经理审批"
        activiti:candidateGroups="generalManager">
        // 省略表单字段
    </userTask>
    <conditionExpression xsi:type="tFormalExpression">
        <![CDATA[${amountMoney >= 10000}]]> #1
    </conditionExpression>
    </sequenceFlow>
    <userTask id="pay" name="出纳付款"
        activiti:candidateGroups="cashier">
        // 省略表单字段
    </userTask>
    <endEvent id="errorendevent1" name="TerminateEndEvent">
        <errorEventDefinition errorRef="PAYMENT_REJECT"/> #2
    </endEvent>
    <sequenceFlow id="flow18" name="金额 < 1万" sourceRef="exclusivegateway2" targetRef="pay">
        <conditionExpression xsi:type="tFormalExpression">
            <![CDATA[${amountMoney < 10000}]]> #3
        </conditionExpression>
    </sequenceFlow>
    <userTask id="treasurerAudit" name="财务审批"
        activiti:candidateGroups="treasurer">
        // 省略表单字段
    </userTask>
    <endEvent id="errorendevent2" name="End">
        <errorEventDefinition errorRef="PAYMENT_REJECT" /> #4
    </endEvent>
</subProcess>
```

以上代码列出了部分关键代码，省略了表单字段部分：#1和#3处利用主流程中的变量`amountMoney`作为条件表达式的一部分；#2和#4处定义了一个异常结束事件，对应图10-1中箭头标记处。

在主流程中附加在付费子流程的异常边界事件的定义如下：

```
<boundaryEvent id="boundaryerror1" name="Error"
    attachedToRef="subprocessPay">
    <errorEventDefinition errorRef="PAYMENT_REJECT" />
</boundaryEvent>
<sequenceFlow id="flow21" name="捕获子流程的异常事件"
    sourceRef="boundaryerror1" targetRef="modifyApply" />
```

从上面的代码可以看出：捕获到编码为“`PAYMENT_REJECT`”的异常后把输出流指向ID为“`modifyApply`”的活动，即“调整申请”。

对于流程的单元测试，读者可以参考本书配套资源文件的本章目录下的`PurchaseSub-ProcessTest.java`类。

10.1.2 流程办理

 **提示** 启动本章的示例后会自动部署流程定义，读者可以参考`applicationContext.xml`中属性“`deploymentResources`”的配置。

由于本例的用户任务涉及多个不同的角色处理，为了方便演示并接近真实需求添加了几个角色和用户，如表10-1和表10-2所示。

表10-1 新添加的角色

角 色 名 称	角 色 ID
财务人员	treasurer
出纳员	cashier
后勤人员	supportCrew

表10-2 新添加的用户

用 户 名 称	用 户 ID	分 配 角 色
Tony Zhang	treasurer	treasurer, cashier
Thomas Wang	thomas	supportCrew

以“henry”用户登录系统，然后启动一个新的“办公用品采购”流程实例，如图10-2所示。

更换用户“kermit”在任务列表中显示一条待办任务，签收并单击“办理”按钮后打开办理页面，如图10-3所示。

任务办理—[领导审批]，流程定义ID：[purchase-subprocess:1:110]

申请人:

henry

到货期限:

2013-01-27

物品清单:

1. MacBook Pro一台
2. 27寸显示器一台

是否同意:

同意

返回列表

完成任务

图10-3 领导审批表单

更换用户“thomas”在任务列表中显示一条待办任务，签收并单击“办理”按钮打开办理页面，如图10-4所示。单击“完成任务”之后完成当前任务，同时引擎自动启动一个“付费子流程”的流程实例，从图10-5中的数据库记录可以清楚地看出两条流程实例由字段“PARENT_ID”维护关系，10.1.3节会详细说明。

任务办理—[联系供货方]，流程定义ID：[purchase-subprocess:1:110]

申请人:

henry

到货期限:

2013-01-27

物品清单:

1. MacBook Pro一台
2. 27寸显示器一台

总金额:

22300.0

供货方:

苹果公司

开户行:

中国工商银行

银行账号:

6222972347027340242

预计交货日期:

2013-01-25

只读字段

需要由后勤人员填写

返回列表

完成任务

图10-4 用户任务“联系供货方”表单

SELECT * FROM ACT_RU_EXECUTION;

ID_	REV_	PROC_INST_ID_	BUSINESS_KEY_	PARENT_ID_	PROC_DEF_ID_	SUPER_EXEC_	ACT_ID_	IS_ACTIVE_
141	1	114	null	114	purchase-subprocess:1:110	null	treasurerAudit	TRUE
114	3	114	null	null	purchase-subprocess:1:110	null	null	FALSE

(2 rows, 2 ms)

图10-5 启动了付费子流程的数据库记录

对于单个流程来说，默认表“ACT_RU_EXECUTION”的“ID_”字段与“PROC_INST_ID_”字段的值是相同的，如果一个流程包含子流程、调用活动、多实例等活动类型，由主流程触发（自动启动）的Execution对象是附属于主流程的，通过设置Execution记录的“PARENT_ID_”字段的值为主流程实例的ID作为维护关系的条件。

更换用户“tony”在任务列表中显示一条名称为“财务审批”的待办任务，签收并单击“办理”按钮打开办理页面，如图10-6所示。

任务办理—[财务审批]，流程定义ID:

申请人:

henry

用途:

1. MacBook Pro一台
2. 27寸显示器一台

总金额:

22300.0

是否同意:

同意

返回列表

完成任务

图10-6 用户任务“财务审批”表单

由于申请的金額超过了1万元，因此在tony办理完任务“财务审批”之后任务流转至“总经理审批”，更换用户“bill”后显示的任务列表如图10-7所示。

任务ID	任务名称	流程实例ID	流程定义ID	任务创建时间	办理人	操作
151	总经理审批	114	purchase-subprocess:1:110	Sun Jan 27 18:58:14 CST 2013	bill	

图10-7 总经理的待办任务

在“总经理审批”任务办理完成之后再次以“tony”身份登录系统办理用户任务“出纳付款”，如图10-8所示。在办理完任务“出纳付款”之后任务流转至“收货确认”（自动设置任务的办理人），如图10-9所示。

任务办理—[出纳付款]，流程定义ID：

申请人：

henry

用途：

1. MacBook Pro一台
2. 27寸显示器一台

受款方：

苹果公司

开户行：

中国工商银行

银行账号：

6222972347027340242

返回列表

完成任务

图10-8 用户任务“出纳付款”表单

以上流程的办理是在假设所有审批节点都通过的情况下进行的，读者可以体验一下“财务审批”或“总经理审批”不通过的情况（在是否同意字段选择“驳回”）：在选择了“驳回”选项后完成任务会执行异常结束事件，捕获到异常事件之后任务流转至“调整申请”。

任务ID	任务名称	流程实例ID	流程定义ID	任务创建时间	办理人	操作
161	收货确认	114	purchase-subprocess:1:110	Sun Jan 27 19:04:02 CST 2013	henry	

图10-9 待办任务“收货确认”

10.1.3 分析流程数据

10.1.1节用XML的方式说明了子流程如何定义，10.1.2节用实例的方式展示了包含子流程的流程的流转过程，本节将从数据库角度分析，解释为什么子流程可以获取主流程的流程变量。

图10-5展示的是运行时执行实例（Execution）表ACT_RU_EXECUTION的数据，从图中可以看出其流程实例ID均为114，但是ID_的值不同（分别为114，141）。114既是流程实例ID又是一个执行实例ID，也可以理解为第一次启动流程时的执行实例ID即为流程实例ID；但是ID_为141的记录仅仅是一个执行实例，当一个流程有多个执行实例时，每个执行实例的“PROC_INST_ID_”和“PARENT_ID_”字段的值均为114，因此就可以把一些了执行实例串联起来。

 **提示** 流程实例（ProcessInstance）与执行实例（Execution）是一对多的关系。

我们知道Activiti引擎把数据划分为运行（runtime）和历史（history）两个级别，图10-5展示的数据属于运行时，而在历史流程实例表中的数据则不同，仅仅只有一条流程实例记录，如图10-10所示。

SELECT * FROM ACT_HI_PROCINST;									
ID_	PROC_INST_ID_	BUSINESS_KEY_	PROC_DEF_ID_	START_TIME_	END_TIME_	DURATION_	START_USER_ID_	START_ACT_ID_	END_ACT_ID_
114	114	null	purchase-subprocess:1:110	2013-01-27 17:16:36.806	2013-01-27 19:09:14.817	6758011	henry	startevent1	endevent3

图10-10 历史流程实例数据

在10.1.1节中，子流程的流程定义中包含了一个排他分支用于判断金额是否超过1万元，而判断条件是否成立的表达式使用的是主流程的表单字段“amountMoney”，我们没有在任何地方为子流程设置该变量的值，为什么子流程可以使用呢？图10-11用实际的数据给出了很好的解释。

简单来说，子流程可以“共享”主流程的所有变量，而且子流程设置的变量也以主流程的执行实例ID作为依据，所以子流程在获取数据时使用主流程的执行实例ID（子流程对应的执行实例的PARENT_ID_字段值）就可以获取到主流程的变量。

SELECT * FROM ACT_HI_VARINST;										
ID_	PROC_INST_ID_	EXECUTION_ID_	TASK_ID_	NAME_	VAR_TYPE_	REV_	BYTEARRAY_ID_	DOUBLE_	LONG_	TEXT_
115	114	114	null	applyUserId	string	0	null	null	null	henry
120	114	114	null	dueDate	date	0	null	null	1359216000000	null
121	114	114	null	listing	string	0	null	null	null	1. MacBook Pro一台 2. 27寸显示器一台
122	114	114	null	amountMoney	double	0	null	22300.0	null	null
127	114	114	null	deptLeaderApproved	string	0	null	null	null	true
136	114	114	null	supplier	string	0	null	null	null	苹果公司
137	114	114	null	bankName	string	0	null	null	null	中国工商银行
138	114	114	null	bankAccount	string	0	null	null	null	6222972347027340242
139	114	114	null	planDate	date	0	null	null	1359043200000	null
140	114	114	null	usage	string	0	null	null	null	1. MacBook Pro一台 2. 27寸显示器一台
147	114	114	null	treasurerApproved	string	0	null	null	null	true
154	114	114	null	generaManagerApproved	string	0	null	null	null	true

图10-11 流程变量表查询结果

10.2 调用活动

如果说子流程是一个类中的一个private方法，只能被当前类调用，那么调用活动就更上一层，把一个通用的方法提取到一个独立的类中供所有的类使用。

一般来说每个公司都有一套完整的付款流程，而且适用于不同的业务，所以把付费流程设计为一个独立且通用的流程是很有必要的。

10.2.1 流程定义

在了解了子流程与调用活动的区别后我们利用10.1节的例子把一个流程分离为两个独立的流程，以达到付费流程的通用性目的。图10-12列出了拆分后的两个独立流程定义的文件列表。

Name	
	payment.bpmn
	payment.png
	payment.zip
	purchase-callactivity.bpmn
	purchase-callactivity.png
	purchase-callactivity.zip

图10-12 拆分流程后的文件列表

其中payment.bpmn为拆分后独立的付费流程，purchase-callactivity.bpmn为主流程。

图10-13展示了办公用品采购的主流程（purchase-callactivity.bpmn），图10-14展示了抽取后独立的付费流程（payment.bpmn）。

与子流程不同的是：调用活动是以“引用”的方式把一个流程作为主流程的一个活动，当输出流到达调用活动时引擎会自动启动一个独立的流程实例，并把主流程的执行实例作为上级执行实例（Super Execution）。

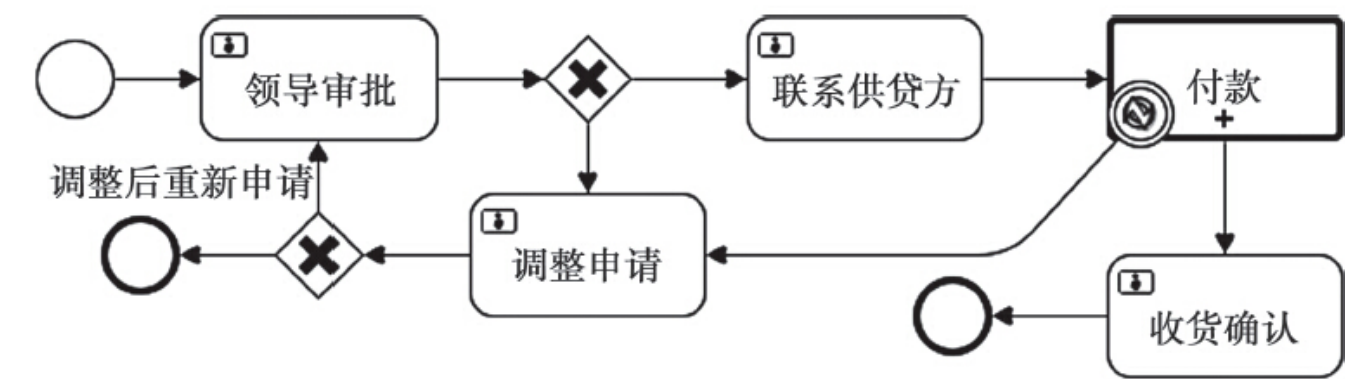


图10-13 办公用品采购主流程

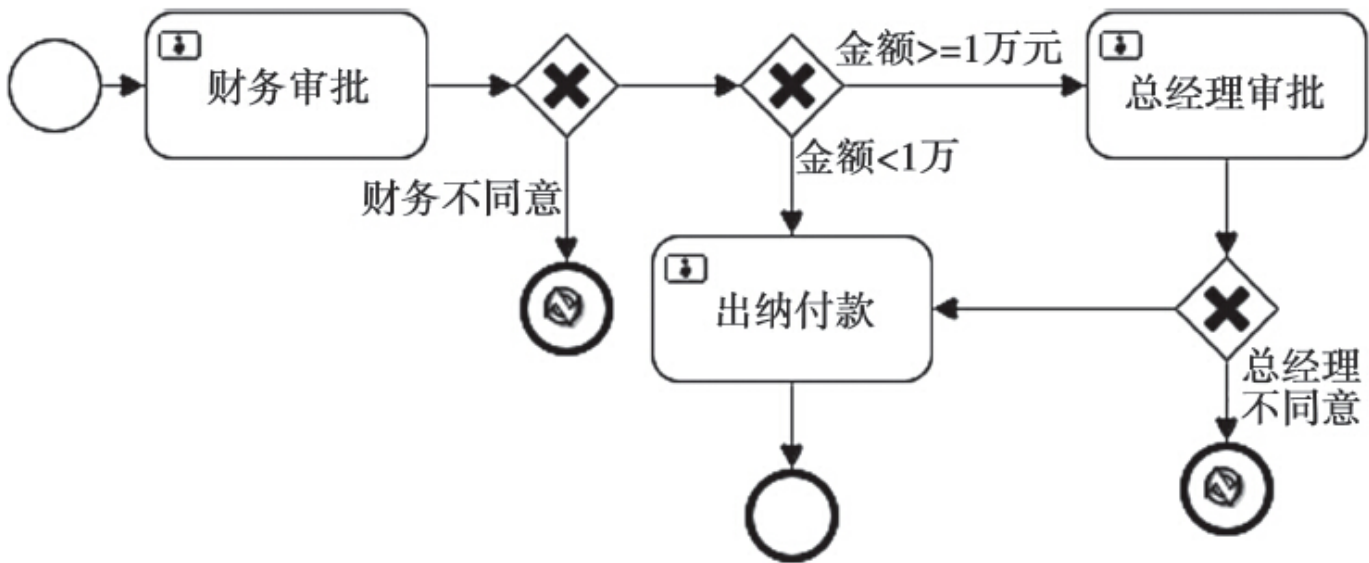


图10-14 独立通用的付费流程

本例的业务流程和10.1.1中一致，不再做重复的讲解，仅针对子流程与调用活动的XML定义做出比较。

```
<process id="purchase-callactivity" isExecutable="true"
  name="办公用品采购--callactivity方式">
  http://www.hzcourse.com/resource/readBook?path=/openresources/teach_ebook/uncompressed/15030/OEBPS/Text/...
  <callActivity id="callactivity-payment"
    name="付款" calledElement="payment">
    <extensionElements>
      <activiti:in source="applyUserId" target="applyUserId"/>
      <activiti:in source="listing" target="usage"/>
      <activiti:in source="amountMoney" target="amountMoney"/>
    </extensionElements>
  </callActivity>
  http://www.hzcourse.com/resource/readBook?path=/openresources/teach_ebook/uncompressed/15030/OEBPS/Text/...
</process>
```

使用callActivity代替了subProcess，使用属性calledElement的值定义了外部流程的流程ID，使用activiti:in标签定义了两个输入类型的变量，这样独立的付款流程就可以使用target属性值作为变量名获取变量，因为调用活动不能读取调用方的变量，必须通过activiti:in显示的指定。

10.2.2 单元测试

对于10.1.1节的流程定义，可以运行PurchaseSubProcessTest.java单元测试来模拟、验证流程的运行过程和结果；对于10.2.1节中调用活动版本的办公用品采购流程，可以运行单元测试类PurchaseCallActivityTest.java来模拟、验证流程的运行过程和结果。

本节的主要是要指出两种测试的不同。

1. 流程部署

对10.1.1节中流程定义进行测试，只需要部署一个流程定义文件即可，代码如下：

```
@Test
@Deployment(resources = {
  "diagrams/chapter10/purchase-subprocess.bpmn" })
public void testAllApproved() throws Exception {
  http://www.hzcourse.com/resource/readBook?path=/openresources/teach_ebook/uncompressed/15030/OEBPS/Text/...
}
```


但是对于10.2.1节的流程定义则不同，需要部署两个流程文件方可正常运行测试，因为在流程执行到“付款”活动时需要调用外部的流程（流程ID为“payment”），代码如下：

```
@Test
@Deployment(resources = {
    "diagrams/chapter10/purchase-callactivity.bpmn",
    "diagrams/chapter10/payment.bpmn" })
public void testAllApproved() throws Exception {
    http://www.hzcourse.com/resource/readBook?path=/openresources/teach_ebook/uncompressed/15030/OEBPS/Text/...
}
```

2.变量获取

从子流程的测试类中获取变量“usage”的代码如下，其中processInstance为主流程的流程实例对象。

```
Execution subExecution =
    runtimeService.createExecutionQuery()
        .processInstanceId(processInstance.getId())
        .activityId("treasurerAudit").singleResult();
assertNotNull(subExecution);
assertEquals(listing, runtimeService
    .getVariable(processInstance.getId(), "usage"));
```

对于10.2.1节中的流程定义，要获取变量“usage”这种方式就行不通了，因为两个流程实例是独立的，不可以相互共享变量，只有子流程才可以如此。具体的代码如下：

```
Execution subExecution =
    runtimeService.createExecutionQuery()
        .processDefinitionKey("payment").singleResult();
assertNotNull(subExecution);
assertEquals(listing, runtimeService
    .getVariable(subExecution.getId(), "usage"));
```

从上面的代码可以看出：获取subExecution对象的方式是根据流程定义的Key属性查询的，而在子流程测试中是根据主流程的执行实例ID获取的。

10.2.3 流程办理

图10-15中有3个流程定义对象，分别是10.1节中包含子流程的办公用品采购，以及本节拆分后得到的两个独立流程定义：办公用品采购——callactivity方式、通用付款流程。

payment:1:15	1	通用付款流程	payment	1	payment.bpmn	payment.png	删除	启动
purchase-callactivity:1:14	1	办公用品采购--callactivity方式	purchase-callactivity	1	purchase-callactivity.bpmn	purchase-callactivity.png	删除	启动
purchase-subprocess:1:16	1	办公用品采购	purchase-subprocess	1	purchase-subprocess.bpmn	purchase-subprocess.png	删除	启动

图10-15 已部署流程定义

因为两个版本的流程定义的流程一致，所以从Web界面看不出有什么不同，读者可以根据10.1.2节的办理步骤完成拆分后的办公用品采购流程。值得提醒的是：当用户任务“联系供货方”完成后注意观察表“ACT_RU_EXECUTION”的数据，流程流转产生的数据将在10.2.4节中详细分析。

另外一点不同是完成任务的Controller方法需要加以改进，如果通过主流程启动的外部流程设置了activiti:initiator属性，则不能获取到当前操作人。对TaskController的completeTask方法进行改进的代码如下，其中加粗部分即为改进之处，这样外部流程就可以获取到触发调用活动的操作人变量。

```
@RequestMapping(value = "task/complete/{taskId}")
public String completeTask(
    @PathVariable("taskId") String taskId,
    HttpServletRequest request) throws Exception {
    // 设置当前操作人，对于调用活动可以获取到当前操作人
    identityService.setAuthenticatedUserId(UserUtil.getUserFromSession(request.getSession()).getId());
    http://www.hzcourse.com/resource/readBook?path=/openresources/teach_ebook/uncompressed/15030/OEBPS/Text/...
}
```

10.2.4 分析流程数据

在子流程节中已经分析了图10-5的数据记录，本节使用的调用活动的版本在运行时产生的数据也与之类似，如图10-16所示，不同的是比图10-5多出了1条记录，这多出来的一条记录是流程运行至“付费”调用活动时产生的。注意第2条记录（ID字段的值为48）的“SUPER_EXEC_”字段的值为46，表明该记录是由ID为46的执行实例创建的。

进一步分析执行实例的状态：流程实例（ID为20）的状态（IS_ACTIVE_字段）为“挂起”，由该流程实例（ID为20）创建的执行实例处于“活动”状态，独立的付费流程实例也处于“活动”状态。

SELECT * FROM ACT_RU_EXECUTION;								
ID_	REV_	PROC_INST_ID_	BUSINESS_KEY_	PARENT_ID_	PROC_DEF_ID_	SUPER_EXEC_	ACT_ID_	IS_ACTIVE_
46	1	20	null	20	purchase-callactivity:1:14	null	callactivity-payment	TRUE
48	1	48	null	null	payment:1:15	46	treasurerAudit	TRUE
20	3	20	null	null	purchase-callactivity:1:14	null	null	FALSE
(3 rows, 5 ms)								

图10-16 办公用品采购的调用活动版本在运行时执行的实例数据记录

对应地，在整个流程实例执行结束之后查询历史流程实例表可以看到如图10-17所示的结果。从记录中可以看出，办公用品采购流程的流程实例ID为120，“ID_”为148的记录是由主流程在执行过程中创建的，通过字段“SUPER_PROCESS_INSTANCE_ID_”的值可以看出，“ID_”为148的记录的“SUPER_PROCESS_INSTANCE_ID_”值为120，以此建立了上下级关系。

SELECT * FROM ACT_HI_PROCINST;											
ID_	PROC_INST_ID_	BUSINESS_KEY_	PROC_DEF_ID_	START_TIME_	END_TIME_	DURATION_	START_USER_ID_	START_ACT_ID_	END_ACT_ID_	SUPER_PROCESS_INSTANCE_ID_	
148	148	null	payment:1:115	2013-01-28 21:51:21.789	2013-02-02 21:30:42.669	430760880	thomas	null	endevent1	120	
120	120	null	purchase-callactivity:1:114	2013-01-28 21:50:40.325	2013-02-02 21:31:05.298	430824973	henry	startevent1	endevent3	null	

图10-17 流程结束之后历史流程实例表的数据记录

和子流程相比较，调用活动除了会创建一个独立并附属于主流程的子流程实例之外，在变量方面还有不同之处：在子流程中设置的流程变量和主流程的变量使用相同的ExecutionId作为保存依据（参考图10-11），而调用活动是一个独立的流程实例，所以在调用活动中设置的变量会和主记录分离开来。从图10-18中可以看出这些变量记录的“EXECUTION_ID_”字段的值有两个，而图10-11中只有一个相同的“EXECUTION_ID_”。

SELECT * FROM ACT_HI_VARINST where PROC_INST_ID_ = '120' or PROC_INST_ID_ = '148' order by ID_;											
ID_	PROC_INST_ID_	EXECUTION_ID_	TASK_ID_	NAME_	VAR_TYPE_	REV_	BYTEARRAY_ID_	DOUBLE_	LONG_	TEXT_	TEXT2_
121	120	120	null	applyUserId	string	0	null	null	null	henry	null
126	120	120	null	dueDate	date	0	null	null	1361894400000	null	null
127	120	120	null	listing	string	0	null	null	null	1. MacBook Pro一台 2. 27寸显示器一台	null
128	120	120	null	amountMoney	double	0	null	22300.0	null	null	null
133	120	120	null	deptLeaderApproved	string	0	null	null	null	true	null
142	120	120	null	supplier	string	0	null	null	null	苹果公司	null
143	120	120	null	bankName	string	0	null	null	null	中国工商银行	null
144	120	120	null	bankAccount	string	0	null	null	null	62232947927342	null
145	120	120	null	planDate	date	0	null	null	1359302400000	null	null
148	148	148	null	applyUserId	string	0	null	null	null	henry	null
149	148	148	null	usage	string	0	null	null	null	1. MacBook Pro一台 2. 27寸显示器一台	null
150	148	148	null	amountMoney	double	0	null	22300.0	null	null	null
202	148	148	null	treasurerApproved	string	0	null	null	null	true	null
209	148	148	null	generalManagerApproved	string	0	null	null	null	true	null

图10-18 办公用品采购——CallActivity版本的变量记录

在图10-18中，字段“EXECUTION_ID_”值为148的变量记录是由独立的付费流程设置的，字段“NAME_”值为applyUserId、usage、amountMoney的记录是在启动付费流程时设置的，而启动时的流程变量一部分是由主流程中的“callActivity”通过“activiti:in”决定的。

10.3 事件子流程

事件子流程（4.5.3节）和子流程类似，把一系列的活动归结到一起处理，不同的是事件子流程不能直接启动，而要“被动”地由其他的事件触发启动。

在10.1.1节的办公用品采购流程中，如果付费子流程未通过财务或总经理审批，则抛出一个异常事件，紧接着异常事件被附加在付费子流程上的异常边界事件捕获，这是处理流程异常的一种方式。

举个例子，对于未通过财务或总经理审批的付费子流程，不用边界异常事件捕获而是在一个子流程（包含一个事件启动流程，可以是异常启动事件、消息启动事件）中进行处理。图10-19展示了办公用品采购的另外一个版本，和10.1.1节的例子的区别体现在对付费子流程中异常情况的处理，图中两个箭头的来源是两个可能抛出异常的“异常抛出事件”，箭头所指的是事件子流程的“异常启动事件”，当异常抛出事件的异常编码能和流程的异常捕获事件（包括异常启动事件、中间异常捕获事件）设置的异常编码匹配就可以被捕获。

如果图10-19中的流程在运行时未通过财务或者总经理审批，则会触发“捕获付费子流程异常”子流程的异常启动事件，从而启动一个子流程处理异常，本例执行一个Java Service任务记录异常信息。

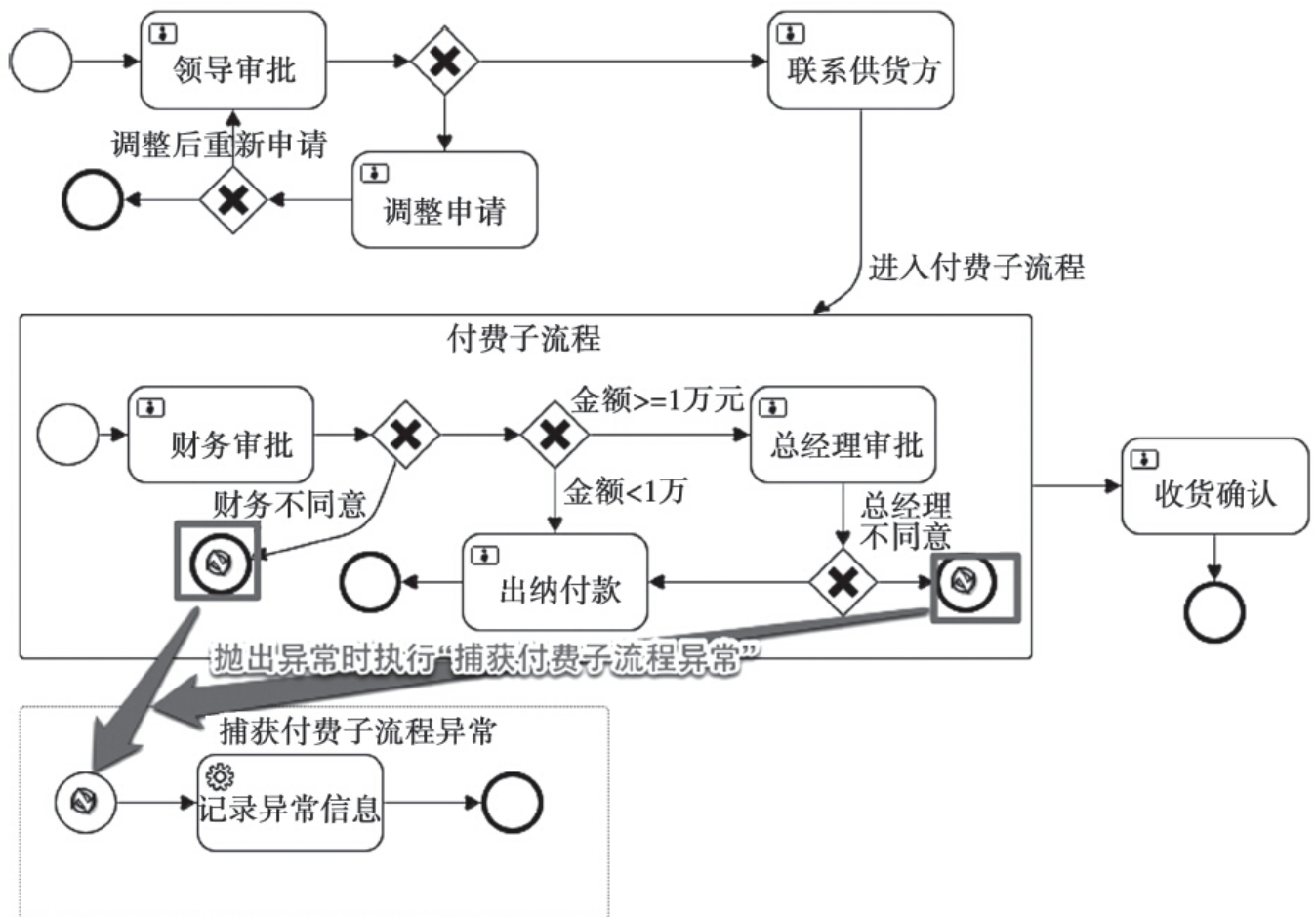


图10-19 包含事件子流程（异常事件）的办公用品采购流程

10.3.1 流程定义

本节主要展示和事件子流程相关的流程定义XML代码，例如当财务或者总经理审批不通过时如何处理以及在事件子流程捕获异常。

和10.1.1节的例子不同，我们在“财务不同意”以及“总经理不同意”输出流上添加了一个执行监听器（下面的代码中的加粗部分）来设置异常信息（保存在一个变量中），对应的XML代码如下：

```
<sequenceFlow id="flow-treasurerAudit" name="财务不同意"
  sourceRef="exclusivegateway-treasurerAudit"
  targetRef="errorrendevent2">
  <extensionElements>
    <activiti:executionListener event="take"
      class="me.kafeitu.activiti.chapter10.listeners
        .HandleErrorInfoForPaymentListener"/>
  </extensionElements>
  <conditionExpression xsi:type="tFormalExpression">
    <![CDATA[ ${treasurerApproved == 'false'} ]]>
  </conditionExpression>
</sequenceFlow>
```

监听器的功能比较简单，设置抛出流程异常的原因，以便事件子流程可以根据原因进行分析处理。监听器的代码如下：

```
public class HandleErrorInfoForPaymentListener
  implements ExecutionListener {
  @Override
  public void notify(DelegateExecution execution)
    throws Exception {
    String activityId = execution.getCurrentActivityId();
    if ("exclusivegateway-treasurerAudit".equals(activityId)) {
      execution.setVariable("message", "财务审批未通过");
    } else if ("exclusivegateway-generalManagerAudit".equals(activityId)) {
      execution.setVariable("message", "总经理审批未通过");
    }
  }
}
```

从上面的代码中可以看出：根据排他分支的ID进行判断，区分审批未通过的来源是财务还是总经理，如此子流程即可获取名称为“message”的变量值分析并处理异常。

定义一个事件子流程的XML代码和普通的子流程类似，唯一不同的是事件子流程在“subProcess”标签上多了一个属性triggeredByEvent=“true”。图10-19中的事件子流程的XML代码如下：

```
<subProcess id="catchErrorForPayment"
  name="捕获付费子流程异常" triggeredByEvent="true">
  <startEvent id="errorstartevent1" name="Error start">
    <errorEventDefinition errorRef="PAYMENT_REJECT"/>
  </startEvent>
  <serviceTask id="recordErrorInfo" name="记录异常信息"
    activiti:expression="${execution.setVariable('ERROR_INFO', message)}">
  </serviceTask>
  <endEvent id="endevent6" name="End">
  </endEvent>
</subProcess>
```

在startEvent中添加元素errorEventDefinition后就可以表示这是一个异常启动事件，一旦当前流程抛出了一个异常且属性errorRef的值匹配，则触发该事件，本例将启动“捕获付费子流程异常”子流程。

10.3.2 单元测试

我们借助代码清单10-2中的单元测试来模拟10.3节开头提到的场景，在付费子流程中抛出一个异常验证是否能够触发事件子流程。

代码清单10-2 验证图10-19中的事件子流程是否执行

```
@ContextConfiguration("classpath:applicationContext-test-chapter10.xml")
public class PurchaseErrorEventSubProcessTest extends SpringActivitiTestCase {
    @Test
    @Deployment(resources = { "diagrams/chapter10/purchase-error-event-subprocess.bpmn" })
    public void testRejectOnTreasurer() throws Exception {
        Map<String, String> properties = new HashMap<String, String>();
        ... // 省略设置变量代码
        identityService.setAuthenticatedUserId("henryyan");
        ... // 省略查询流程定义代码
        ProcessInstance processInstance = formService.submitStartFormData(processDefinition.getId(), properties);
        assertNotNull(processInstance);
        ... // 部门领导审批通过
        ... // 完成任务：联系供货方
        // 子流程-财务审批
        task = taskService.createTaskQuery().taskCandidateGroup("treasurer").singleResult();
        taskService.claim(task.getId(), "kemit");
        properties = new HashMap<String, String>();
        properties.put("treasurerApproved", "false"); // 设置审批未通过
        formService.submitTaskFormData(task.getId(), properties); // #1
        // 验证执行了事件子流程
        HistoricVariableInstance ERROR_INFO = #2-S
        historyService.createHistoricVariableInstanceQuery().variableName("ERROR_INFO").singleResult();
        assertEquals("财务审批未通过", ERROR_INFO.getValue()); // #2-E
    }
}
```

可以在IDE中执行该单元测试，毫无疑问，所有的断言均通过验证。

代码清单10-2的#1处是财务审批任务完成动作，将变量`treasurerApproved`设置为`false`，当任务完成后触发事件子流程的异常启动事件，紧接着执行了Java Service任务“记录异常信息”，若#2处从历史变量中获取名称为“ERROR_INFO”的变量值等于监听器`HandleErrorInfoForPaymentListener.java`设置的变量`message`，则“财务审批未通过”。

10.4 多实例支持

在上一章中详细讲解了多实例的应用，使读者学会了批量任务处理。多实例支持的活动类型有很多，本章介绍的子流程与调用活动可以支持多实例特性。

在实际应用中子流程的多实例应用也比较广泛，例如领导向下属分发任务（注意这里的任务不是一个普通的用户任务，而是一个子流程或者调用活动），并且要求每一个子流程中的某个任务自动分配给多实例的参与者。图10-20展示了任务分发的流程图。

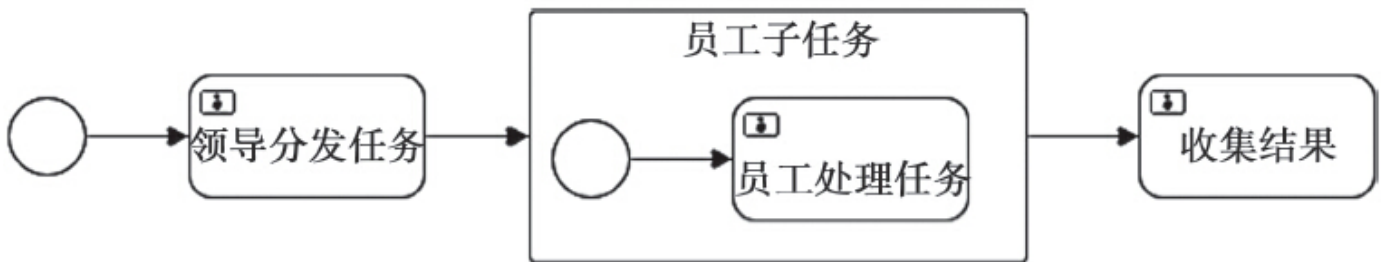


图10-20 任务分配流程图（子流程多实例）

图10-20中的“员工子任务”子流程启用了多实例特性支持，以并行方式执行；“员工子任务”子流程实例个数由主流程的“领导分发任务”设置的一个集合“users”决定，将用户任务“员工处理任务”的`activiti:assignee`属性设置为“\${user}”，如此在完成“领导分发任务”后就可以创建多个“员工子任务”实例，并且用户任务“员工处理任务”的`activiti:assignee`属性自动分配。和“员工子任务”子流程有关的XML代码如下：

```
<subProcess id="subprocess1" name="员工子任务">
    <multiInstanceLoopCharacteristics isSequential="false"
        activiti:collection="${users}"
        activiti:elementVariable="user"/>
    <startEvent id="startevent2" name="Start"
        activiti:initiator="subProcessStartUser"/></startEvent>
    <userTask id="usertask1" name="员工处理任务"
        activiti:assignee="${user}"></userTask>
</subProcess>
```

下面通过一个简单的单元测试验证结果是否和预期一致，见代码清单10-3。

代码清单10-3 子流程多实例单元测试

```
public class MultiInstanceForSubprocessTest extends SpringActivitiTestCase {
    @Test
    @Deployment(resources = { "diagrams/chapter10/multiinstance-for-subprocess.bpmn" })
    public void testOne() throws Exception {
        ProcessInstance processInstance = runtimeService
            .startProcessInstanceByKey("multiinstance-for-subprocess");
        Task task = taskService.createTaskQuery()
            .taskCandidateGroup("deptLeader").singleResult();
        taskService.claim(task.getId(), "bill");
        // 完成任务“领导分发任务”
        Map<String, Object> variables = new HashMap<String, Object>();
        List<String> users = Arrays.asList("user1", "user2", "user3");
        variables.put("users", users);
        taskService.complete(task.getId(), variables);
        // 根据用户集合循环判断，是否每一个用户都有一个待办任务
        for (String user : users) {
            long count = taskService.createTaskQuery().taskAssignee(user).count();
            assertEquals(1, count);
        }
    }
}
```

代码清单10-3验证了我们的假设，结果是用户集合`users`中的每一个用户都有一个待办任务。

此例在日常办公场景中出现，在代码清单10-3中通过设置一个集合设置用户的数量，在实际应用中可以使用交互式的组织架构数选择某些用户决定子流程由哪些用户参与。

如果把图10-20的子流程更换成调用活动也是一样，读者可以动手实验一下。

10.5 本章小结

本章重点讲解了子流程与调用活动，两者有着相似的功能，把一些活动聚集起来处理，但是两者在引擎的实现上有着很大的差别，本章不仅从流程设计上进行了分析，还对引擎的数据库记录做了深入的讲解。

调用活动的目的是重用，当多个流程中出现了重复活动集合时就需要对这一部分重复活动集合进行抽取并设计一个通用的流程，在使用方（主流程）添加一个调用活动引用通用流程即可，免除冗余的设计也避免了通用部分的更改不能和其他流程同步的问题。

子流程则不然，它没有像调用活动那样的通用性，它可以把某些任务归类处理，例如把一个流程划分为多个阶段进行处理（阶段1、阶段2、.....）。

事件子流程可以说是普通子流程的一个特性分支，它只能被事件驱动（异常启动事件、消息启动事件、信号启动事件等），通常用于处理当前流程执行过程中抛出的事件，只要抛出的事件和事件子流程中的启动事件匹配就可以启动一个事件子流程进行处理。

综上所述，有了对子流程及调用活动的深入理解才能灵活运用它们，才能根据需求对号入座。

第11章 事件

第4章介绍了很多目前Activiti支持的事件类型，有启动事件、结束事件、边界事件、中间抛出事件，实战篇的其他章也陆续涉及了一些事件的应用，本章将介绍其他未涉及的事件，带领大家学习如何使用事件以及相关注意事项。

本章不涉及Explorer，全部以单元测试的方式讲解。本章的代码位于bpmn20-example项目的me.kafeitu.activiti.chapter11目录中。

11.1 启动事件

启动事件应该是最熟悉不过的一种事件类型了。每个流程都需要从一个启动事件开始，根据不同的需求可以选择使用特殊功能的启动事件，例如定时启动、异常启动、消息启动等。

在第10章中涉及了异常启动事件，这里补充一下：异常启动事件不能用于主流程，必须嵌入到事件子流程中。

下面分别介绍几种不同的启动事件。

11.1.1 定时启动事件

在8.2.2节的定时发送邮件例子中已经提到了定时边界事件，在BPMN 2.0规范中所有的定时相关的定义都是相同的，定时标签timerEventDefinition总是被嵌套在其他的活动内，在定时发送邮件例子中定时标签嵌套在边界事件boundaryEvent中，所以称之为定时边界事件。

如果定时标签嵌套在启动事件startEvent内，就构成了一个定时启动事件，还可以用相同的方式把其他的事件嵌入到startEvent中，例如异常启动事件、消息启动事件、信号启动事件等。

图11-1展示的是一个用定时启动事件引导的流程，这个流程演示了如何定义以及触发定时启动事件。

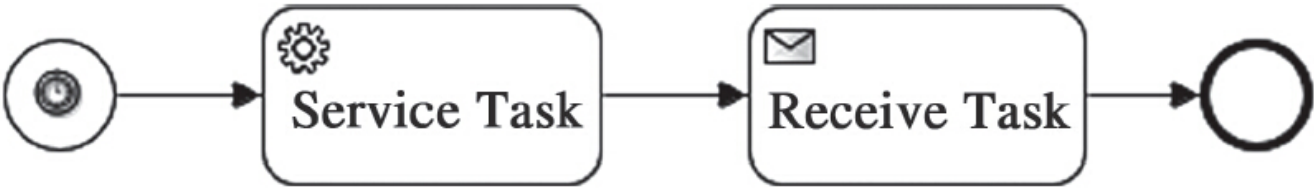


图11-1 包含定时启动事件的流程

在Activiti Designer中，从右侧拖动“TimeStartEvent”到工作区，单击事件的图标在“Properties”视图中可以看到如图11-2所示的设置。


```
@Deployment(resources = "chapter11/timerEvent/timerStartEvent.bpmn")
public void testTriggerManual() throws Exception {
    JobQuery jobQuery = managementService.createJobQuery();
    assertEquals(1, jobQuery.count());
    // 手动触发作业的执行
    Job job = jobQuery.singleResult(); // 获取单个作业对象
    managementService.executeJob(job.getId()); // 手动执行作业
    assertEquals(0, jobQuery.count());
    // 检查是否启动了流程实例
    long count = runtimeService.createProcessInstanceQuery()
        .processDefinitionKey("timerStartEvent").count();
    assertEquals(1, count);
}
```

代码清单11-2和代码清单11-1大致相同，唯一不同的是把模拟5分钟的代码更换成通过API手动调用执行作业，这样不管作业计划在何时执行`executeJob()`方法都会立即执行作业。

代码清单11-1和代码清单11-2演示的都是在部署流程5分钟之后执行作业，也可以参考表4-2设置不同的参数来决定作业如何执行，例如定时一次性（具体到某一天某一秒）、周期性执行。

可以把timerStartEvent.bpmn文件部署到Activiti Explorer中，然后在“Manage”→“Database”视图中查看ACT_RU_JOB表的数据，如图11-4所示（因为排版问题分割成上下两部分）。

 ACT_RU_JOB								
ID_	REV_	TYPE_	LOCK_EXP_TIME_	LOCK_OWNER_	EXCLUSIVE_	EXECUTION_ID_	PROCESS_INSTANCE_ID_	RETRIES_
37	1	timer			true			3

EXCEPTION_STACK_ID_	EXCEPTION_MSG_	DUEDATE_	REPEAT_	HANDLER_TYPE_	HANDLER_CFG_
		2013-02-25 23:01:40.695		timer-start-event	timerStartEvent

图11-4 ACT_RU_JOB表的数据

表11-1列出了ACT_RU_JOB表的一些关键字段的含义。

表11-1 ACT_RU_JOB表的主要字段说明

字段名称	属性说明
TYPE_	作业类型，有 message 和 timer 两种类型
RETRIES_	触发失败后最大重试次数，也可以通过调用 ManagementService#setJobRetries 方法设置
DUEDATE_	执行时间。引擎正是将此字段与当前时间进行比较，相同或已超过则执行作业
HANDLER_TYPE_	作业的处理类型。对于每一种类型都有对应的处理器（handler），目前引擎支持的类型有： ☐ event：事件 ☐ timer-start-event：定时启动事件 ☐ timer-transition：边界定时事件

（续）

字段名称	属性说明
HANDLER_TYPE_	☐ timer-intermediate-transition：定时中间事件 ☐ async-continuation：异步执行的作业
HANDLER_CFG_	作业的目标，和 HANDLER_TYPE_ 联合起来决定执行哪个流程定义的哪个作业处理类型

11.1.2 消息启动事件

消息启动事件和第10章介绍的事件子流程中涉及的异常启动事件类似，都是通过一个特定的消息标志触发事件。一个简单的消息启动事件引导的流程如图11-5所示。

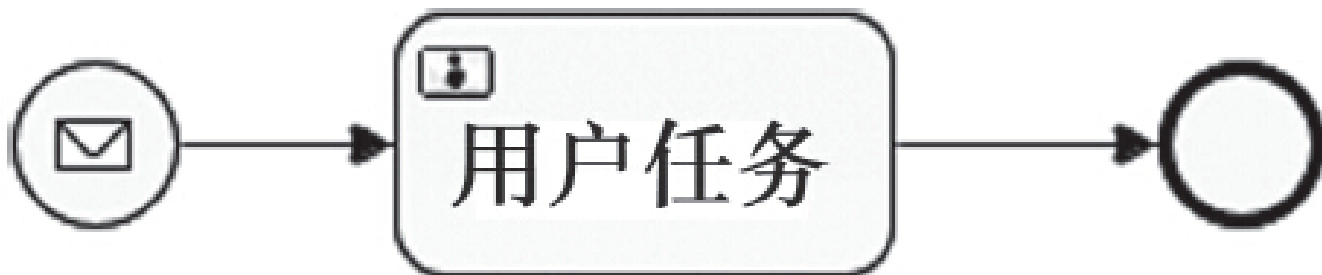


图11-5 消息启动事件流程图示例

使用Activiti Designer设计消息启动事件的方式与设计异常启动事件不同，对于异常启动事件，只要选择后设置“ERROR_CODE”就可以了，而对于消息类型事件，需要先在流程中预设一个消息，之后才能在消息类型的事件上进行配置（下拉框形式选择）。

单击流程设计工作区的空白处（不要选择任何元素），单击“Properties”标签页的“Messages”子标签可以看到如图11-6所示的界面。

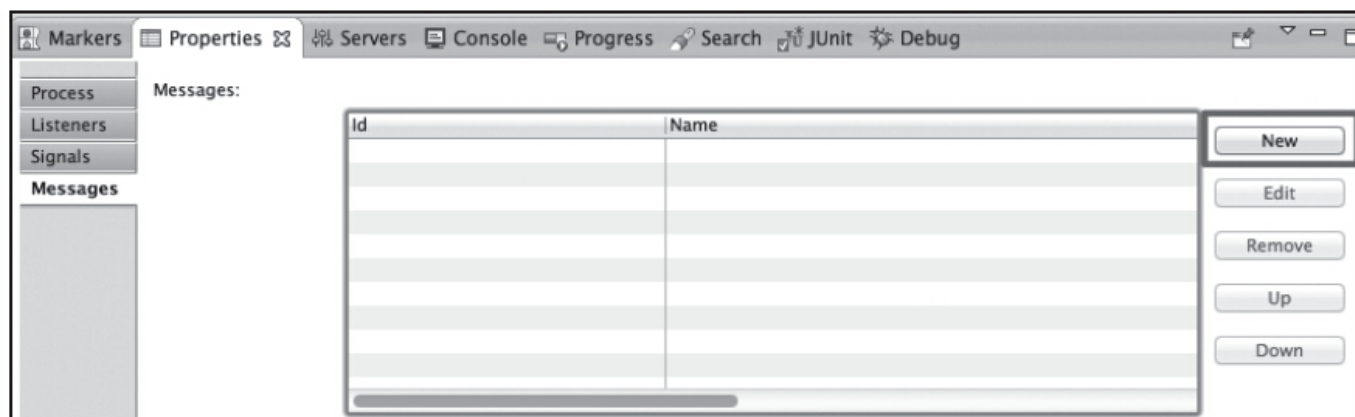


图11-6 流程消息定义界面

单击图11-6的“New”按钮会弹出图11-7的对话框，填写“Id”和“Name”属性。

图11-7 添加消息对话框

单击图11-7的“OK”按钮即可添加一个消息至流程，对应地，设计器会在XML中添加一个“<message.../>”标签。成功添加消息定义之后即可看到如图11-8所示的列表。

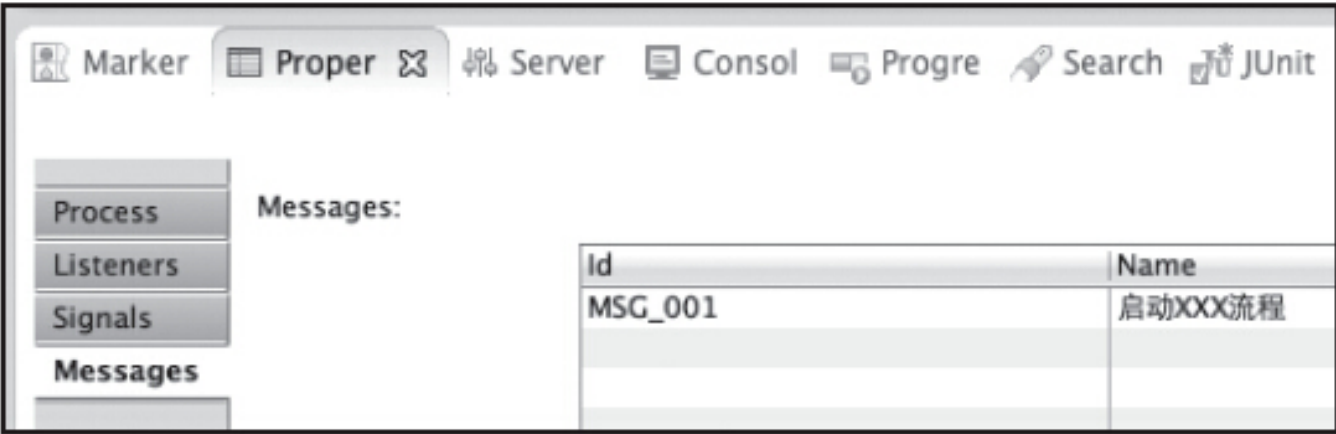


图11-8 消息列表

完成定义之后单击消息启动事件，在“Message”视图可以看到如图11-9所示的界面。单击“Message Ref Id”后面的下拉框可以选择刚刚定义的消息，即图11-8列表中的所有消息，最后选择“MSG_001/启动XXX流程”就完成了消息启动事件的配置。

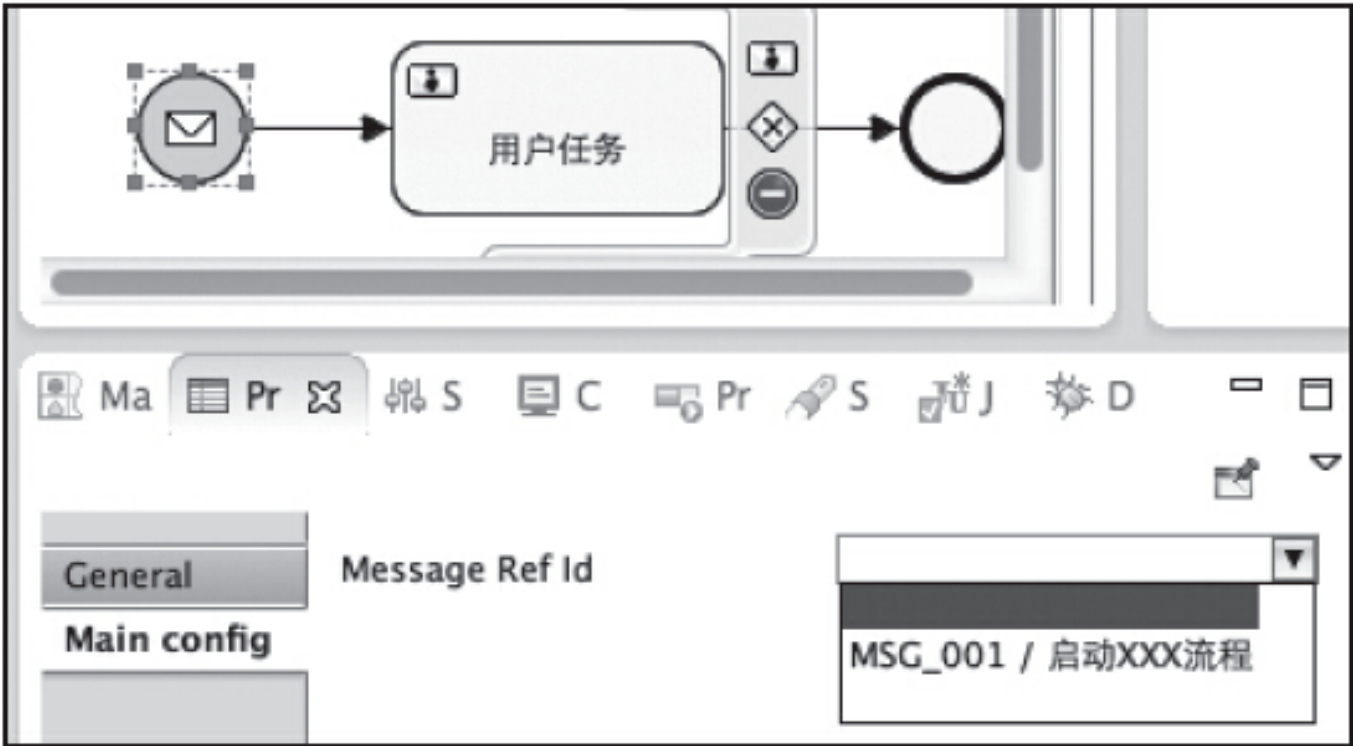


图11-9 为消息启动事件设置消息类型

完成了流程设计之后Activiti Designer会生成XML内容，如代码清单11-3所示。

代码清单11-3 图11-5的XML代码

```
<definitions http://www.hzcourse.com/resource/readBook?path=/openresources/teach_ebook/uncompressed/15030/OEBPS/Text/...>
  <message id="MSG_001" name="启动XXX流程"></message>
  <process id="messageStartEvent" name="messageStartEvent" isExecutable="true">
    <startEvent id="messageStartEvent1" name="Start">
      <messageEventDefinition messageRef="MSG_001"></messageEventDefinition>
    </startEvent>
    http://www.hzcourse.com/resource/readBook?path=/openresources/teach_ebook/uncompressed/15030/OEBPS/Text/...
  </process>
</definitions>
```

代码清单11-3中加粗部分的XML代码就是Activiti Designer生成的，其中<message/>的“id”属性的作用是在流程定义文件内部引用，而“name”属性才是启动流程时的参数。以消息启动事件引导的流程启动方式和之前介绍的启动流程的API有所不同。代码清单11-4列出了通过消息名称启动流程实例的方法。

代码清单11-4 启动消息启动事件引导的流程

```
public class MessageEventTest extends PluggableActivitiTestCase {
    @Deployment(resources = "chapter11/messageEvent/messageStartEvent.bpmn")
    public void testMessageEvent() throws Exception {
        ProcessInstance pi = runtimeService.startProcessInstanceByMessage("启动XXX流程");
        assertNotNull(pi);
    }
}
```

是不是以消息启动事件引导的流程必须使用代码清单11-4的方式启动呢？也不尽然，也可以使用空启动事件的启动方式启动流程，当然这种方式对流程定义有一定的限制：流程中只能包含一个消息启动事件。例如，代码清单11-5通过startProcessInstanceByKey()方法来启动一个流程实例。也可以在查询到流程定义ID之后调用startProcessInstanceById()方法启动流程。

代码清单11-5 通过流程ID启动流程

```
public void testStartMessageEventByKey() throws Exception {
    ProcessInstance pi = runtimeService.startProcessInstanceByKey("messageStartEvent");
    assertNotNull(pi);
}
```

我们知道，定时启动事件可以按照预设时间启动是因为引擎不断刷新数据库表ACT_RU_JOB的记录，根据时间匹配作业，命中之后就执行作业。对消息也采用类似的机制，在部署流程之后引擎会在初始化中处理消息事件，把消息的类型注册到数据库（插入消息记录到表ACT_RU_EVENT_SUBSCR），在流程执行过程中遇到了消息类型事件或通过API触发消息事件会从该表读取数据，并且根据消息的属性调用消息处理器。

图11-10展示了表ACT_RU_EVENT_SUBSCR的表记录。

ACT_RU_EVENT_SUBSCR							
ID_	REV_	EVENT_TYPE_	EVENT_NAME_	EXECUTION_ID_	PROC_INST_ID_	ACTIVITY_ID_	CONFIGURATION_
62	1	message	启动XXX流程			messagestartevent1	messageStartEvent:1:61

图11-10 表ACT_RU_EVENT_SUBSCR

下面列出每个字段的含义，以帮助读者从中理解引擎处理消息事件的原理。

- EVENT_TYPE_: 事件类型。message表示消息类型。
- EVENT_NAME_: 消息名称。触发消息时引擎会根据消息名称参数和这个字段进行匹配。
- EXECUTION_ID_: 实例执行ID。如果一个流程以消息启动事件引导，则此字段为空，如果在流程运行中注册了消息事件，则此字段保存流程执行ID。
- PROC_INST_ID_: 流程实例ID。同EXECUTION_ID_，只不过保存的是流程实例ID（与EXECUTION是一对多的关系）。
- ACTIVITY_ID_: 活动ID。如果根据字段EVENT_NAME_匹配到消息就应该触发哪个活动。
- CONFIGURATION_: 消息的配置信息。对于消息类型来说这里保存的是流程定义ID。

根据对字段的解释已经很清楚引擎是如何处理消息事件的了：当我们调用[startProcess-InstanceByMessage](#)方法时，根据匹配规则就能触发消息启动事件，从而启动一个流程实例。

另外字段EXECUTION_ID_与PROC_INST_ID_不为空的情况将在11.3.2节的消息边界事件应用中介绍。

根据消息名称启动流程（代码清单11-3）可能会遇到如下的运行时异常，原因是对应的消息名称不存在。

```
org.activiti.engine.ActivitiException: Cannot start process instance by message: no subscription to message with name '启动付费流程' found.
```

遇到这样的问题我们会想到先从数据库查询是否已经注册过名称为“启动付费流程”的消息，有则启动，没有则提示用户启动流程失败。

不过目前Activiti引擎没有公开提供查询消息的接口，所以我们要利用非公开的查询类查询消息列表，如代码清单11-6所示。

代码清单11-6 查询对应的消息是否存在

```
@Deployment(resources = "chapter11/messageEvent/messageStartEvent.bpmn")
public void testMessageSubscription() throws Exception {
    EventSubscriptionQueryImpl eventSubscriptionQuery = new EventSubscriptionQueryImpl(
        processEngineConfiguration.getCommandExecutor());
    EventSubscriptionEntity subscriptionEntity = eventSubscriptionQuery
        .eventName("启动XXX流程").singleResult();
    assertNotNull(subscriptionEntity);
}
```

11.2 结束事件

终止结束事件（4.1.3节）和空结束事件最大的区别就是前者可以终止整个流程实例的执行，而后者只结束一条输出流的执行。

图11-11所示的流程图包含一个终止结束事件，启动事件后面并行网关的两个输出流分别执行一个子流程和“主任务A”，在“主任务A”完成后终止结束事件将被触发，从而终止流程实例的执行，此时从历史任务中查询到流程实例状态为已完成。

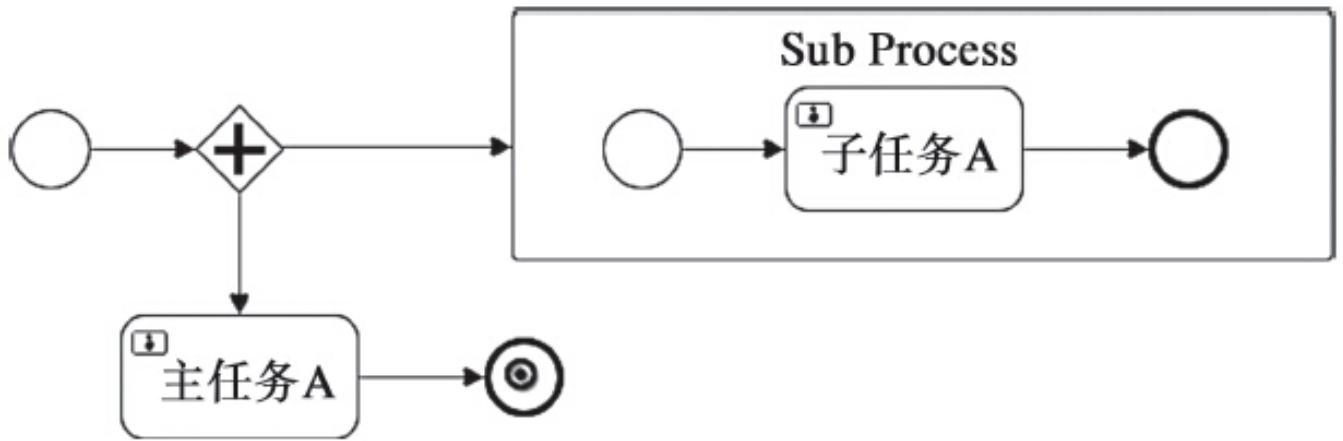


图11-11 包含终止结束事件的流程

通过两个简单的测试可以快速理解终止结束事件的作用，以及对流程数据的影响：第一个单元测试（代码清单11-7）先把子流程执行完成，然后再完成主流程的用户任务“主任务A”；第二个单元测试（代码清单11-8）直接完成“主任务A”结束流程实例的执行，然后观察这两种不同的执行顺序对流程数据的影响。

代码清单11-7 先完成子流程再完成“主任务A”

```
@Deployment("chapter11/terminateEndEvent/terminateEndEventWithSubprocess.bpmn")
public void testOne() throws Exception {
    ProcessInstance processInstance = runtimeService
        .startProcessInstanceByKey("terminateEndEventWithSubprocess");
    assertNotNull(processInstance);
    // 查询名称为“子任务A”的任务
    Task task = taskService.createTaskQuery().taskDefinitionKey("subTask").singleResult();
    taskService.complete(task.getId());
    // 完成主流程用户任务
    task = taskService.createTaskQuery().taskDefinitionKey("masterTask").singleResult();
    taskService.complete(task.getId());
    checkFinished(processInstance);
}
```

代码清单11-7中的checkFinished()方法的代码如下，用于监测流程状态和输出结果。

```
private void checkFinished(ProcessInstance processInstance) {
    // 验证流程已结束
    HistoricProcessInstance hpi = historyService
        .createHistoricProcessInstancesQuery()
        .processInstanceId(processInstance.getId())
        .singleResult();
    assertNotNull(hpi.getEndTime());
    // 查询历史任务
    List<HistoricTaskInstance> list = historyService
        .createHistoricTaskInstancesQuery().list();
    for (HistoricTaskInstance hti : list) {
        System.out.println(hti.getName() + " "
            + hti.getDeleteReason());
    }
    // 流程结束后校验监听器设置的变量
    HistoricVariableInstance vi = historyService
        .createHistoricVariableInstancesQuery()
        .variableName("settedOnEnd").singleResult();
    assertEquals(true, vi.getValue());
}
```

 **说明** 如果主流程设置了“end”监听器，在流程终止时将被调用。

运行完代码清单11-7的单元测试之后控制台会打印如下内容：

```
子任务A    completed
主任务A    completed
```

解释一下输出的“completed”，如果一个任务正常完成（即通过调用taskService.complete()方法），历史任务对象的删除原因（属性deleteReason的值）为“completed”，如果任务被删除，则删除原因为“deleted”。

从代码清单11-7中可能看不出有什么特别的地方，接着来看代码清单11-8的测试代码，直接完成用户任务“主任务A”，再观察数据结果与代码清单11-7的输出结果有何不同。

代码清单11-8 直接完成“主任务A”结束流程

```
public void testSecond() throws Exception {
    ProcessInstance processInstance = runtimeService
        .startProcessInstanceByKey("terminateEndEventWithSubprocess");
    assertNotNull(processInstance);
    // 完成主流程用户任务
    Task task = taskService.createTaskQuery()
        .taskDefinitionKey("masterTask").singleResult();
    taskService.complete(task.getId());
    checkFinished(processInstance);
}
```

先观察输出结果，然后再解释为什么结果和代码清单11-7不同。

```
子任务A    deleted
主任务A    completed
```

“子任务A”的删除原因为“deleted”，与“主任务A”的删除原因不同。刚刚解释了任务被删除的原因为“deleted”，由此可以看出在终止结束事件被执行后所有处于运行中的任务都将被删除（如果有监听器也会被调用）。

11.3 边界事件

边界事件应该是实际应用中比较常用的事件类型。利用边界事件可以以“插件式”为活动添加附件（边界事件总是附加在某一个活动上）事件，比如，在第8章中利用定时边界事件自动发送销假超时提醒邮件，在第10章中利用异常边界事件对付费异常进行处理。

本节内容将深入介绍边界事件的其他特性，并补充讲解前面章节未涉及的其他边界事件。

11.3.1 异常边界事件

在第10章中利用异常边界事件处理付费子流程的异常情况看起来很优雅，也是中规中矩的，为什么这么说呢？因为我们可以让触发异常边界事件更“Geek”。第10章的例子是在付费子流程中利用异常结束事件抛出异常信息，如果一个异常附加在用户任务或者Java Service任务上该如何抛出异常事件呢？

首先来看图11-12所示的流程图，在“自动执行系统任务”上附加了一个异常边界事件，在捕获到异常后输出流指向用户任务“处理异常”，处理完此任务后再次执行“自动执行系统任务”，此时“自动执行系统任务”会执行一个实现了JavaDelegate接口的类，根据设定的业务逻辑决定何时抛出异常。

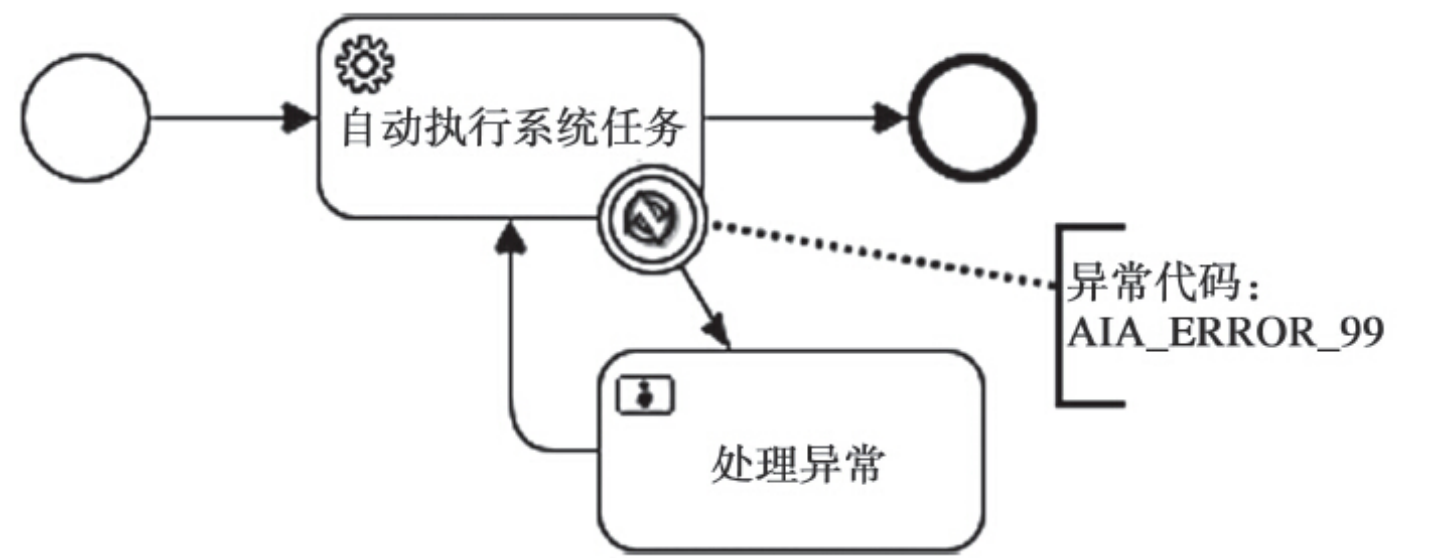


图11-12 手动抛出异常事件

了解了流程处理过程之后再来看看该流程的XML代码，这里只列出了关键部分。

```
<process id="throwErrorManual" name="throwErrorManual" isExecutable="true">
  <serviceTask id="servicetask1" name="自动执行系统任务"
    activiti:class="me.kafeitu.activiti.chapter11
      .listener.ThrowErrorManualService" />
  <boundaryEvent id="boundaryerror1" name="Error"
    attachedToRef="servicetask1">
    <errorEventDefinition errorRef="AIA_ERROR_99"/>
  </boundaryEvent>
</process>
```

从上面的代码清单中可以看出图11-12的“自动执行系统任务”执行一个Java类（见代码清单11-9），附加的异常边界事件捕获的错误代码为“AIA_ERROR_99”。代码清单11-10对本节开始的执行过程进行了模拟与验证。

代码清单11-9 直接完成“主任务A”结束流程

```
public class ThrowErrorManualService implements JavaDelegate {
    @Override
    public void execute(DelegateExecution execution) throws Exception {
        if (execution.getVariable("pass") == null) { // 如果变量pass的值为空就抛出异常
            throw new BpmnError("AIA ERROR 99");
        } // 抛出一个BpmnError对象即可手动抛出异常，从而使异常被附加在任务上的
        // 异常边界事件捕获
    }
}
```

代码清单11-10 手动抛出异常

```
public class ThrowErrorManualTest extends PluggableActivitiTestCase {
    @Deployment(resources = "chapter11/boundaryEvent/throwErrorManual.bpmn")
    public void testThrowErrorManual() throws Exception {
        ProcessInstance processInstance = runtimeService
            .startProcessInstanceByKey("throwErrorManual");
        assertNotNull(processInstance);
        // 流转至用户任务
        Task task = taskService.createTaskQuery().singleResult();
        Map<String, Object> variables = new HashMap<String, Object>();
        variables.put("pass", true); // 完成任务时设置pass变量，使其值不为null
        taskService.complete(task.getId(), variables);
        // 流程执行完成
        long count = historyService.createHistoricProcessInstanceQuery().finished().count();
        assertEquals(1, count);
    }
}
```

结合代码清单11-9和代码清单11-10的单元测试明白了如何手动抛出异常，从而可以根据流程数据处理复杂的业务逻辑。

11.3.2 消息边界事件

消息边界事件与异常边界事件有着类似的触发方式，可以调用RuntimeService的接口触发消息边界事件。

图11-13展示了包含消息边界事件的流程图，附加在用户任务“审核文件”的消息边界事件可以捕获该任务触发（也可以称之为抛出）的消息事件。

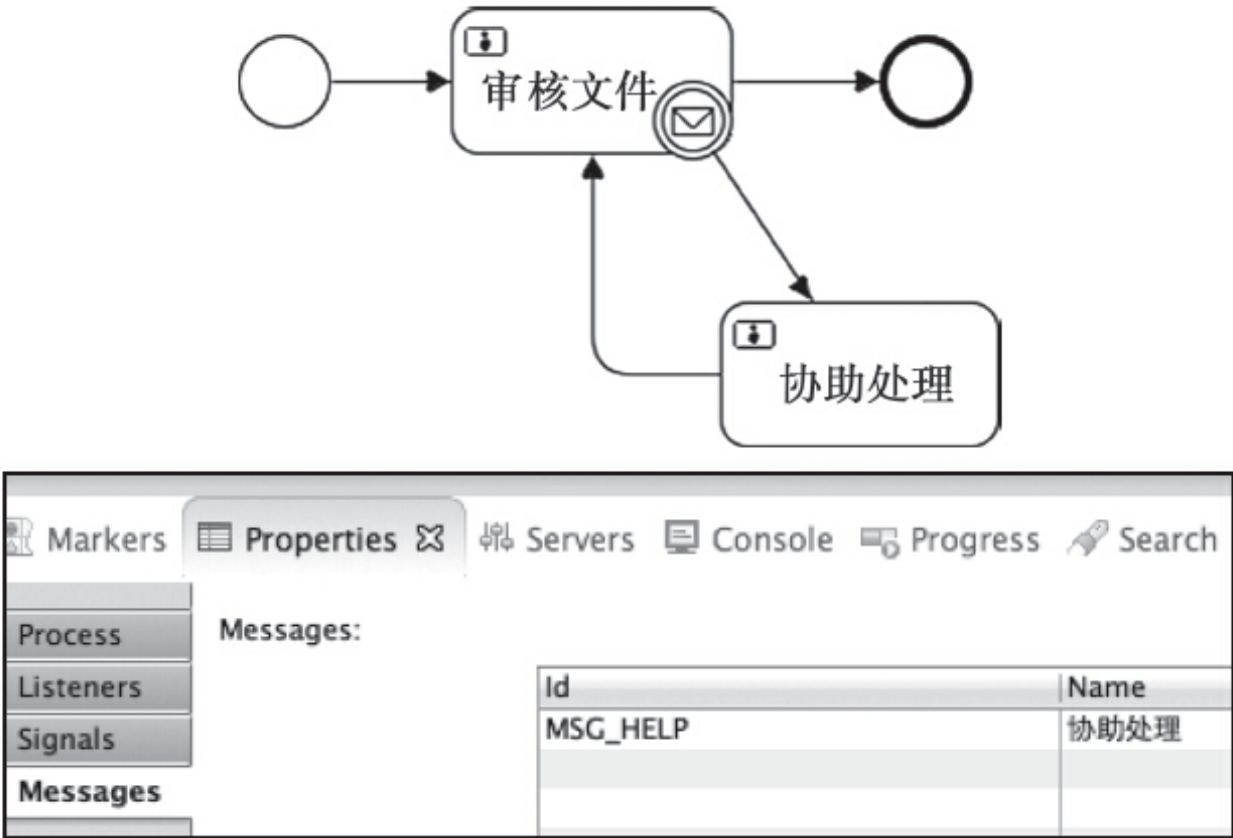


图11-13 包含消息边界事件的文档审批流程图

下面的代码清单列出了图11-13中关键的XML，而消息边界事件监听的消息名称可以参考图11-14的界面进行设置，注意边界事件的“cancelActivity”属性的值为“true”。

```
<message id="MSG_HELP" name="MSG_协助处理"></message>
<process id="messageBoundaryEvent"
  name="messageBoundaryEvent" isExecutable="true">
  <userTask id="usertask1" name="审核文件"></userTask>
  <boundaryEvent id="boundarymessage1" name="Message"
    attachedToRef="usertask1" cancelActivity="true">
    <messageEventDefinition messageRef="MSG_HELP"/>
  </boundaryEvent>
  <userTask id="usertask2" name="协助处理"></userTask>
</process>
```

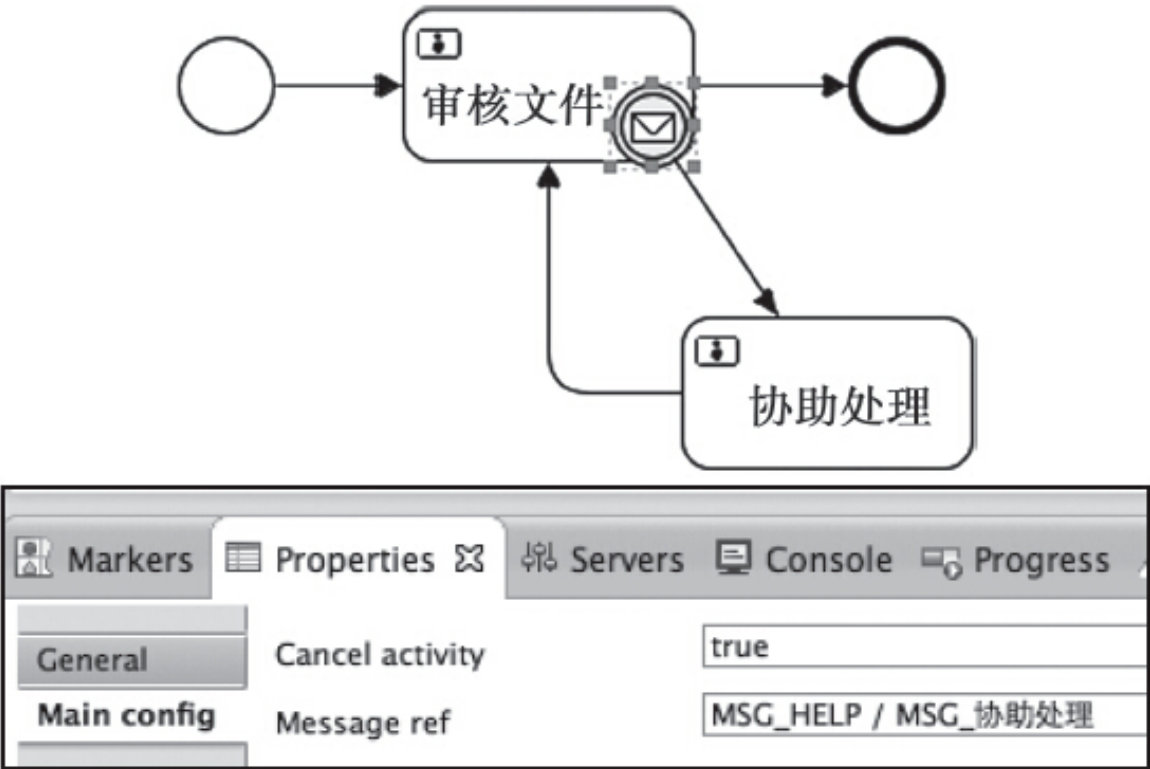


图11-14 设置消息边界事件属性

消息边界事件、信号边界事件与异常边界事件会多出一个“cancelActivity”属性，此属性可以取值true或false。下面对“cancelActivity”属性的不同取值做出了说明。

- true：在消息边界事件触发后取消已注册的消息事件。
- false：在消息边界事件触发后仍然保留已注册的消息事件，可以再次触发。

下面分别介绍cancelActivity属性取不同值时所产生的不同结果。

1.单次触发

图11-13的流程业务逻辑很简单，假设员工需要提交一个文件给领导审核，在启动流程时上传附件启动流程，在领导审核过程中需要他人协助处理，可以通过发送消息的方式把任务转交出去，等处理完再回归到“审核文件”用户任务。

下面通过单元测试的方式模拟这个假设的业务流程，见代码清单11-11所示。

代码清单11-11 消息边界事件单元测试

```
public class ReceiveMessageManualTest extends PluggableActivitiTestCase {
    @Deployment(resources = "chapter11/boundaryEvent/messageBoundaryEvent.bpmn")
    public void testReceiveMessageManual() throws Exception {
        runtimeService.startProcessInstanceByKey("messageBoundaryEvent");
        // 审核文件任务
        Task task = taskService.createTaskQuery().taskName("审核文件").singleResult();
        assertNotNull(task);
        ExecutionQuery executionQuery = runtimeService.createExecutionQuery();    #1-S
        Execution execution = executionQuery
            .messageEventSubscriptionName("MSG_协助处理").singleResult();
        runtimeService.messageEventReceived("MSG_协助处理", execution.getId());    #1-E
        // 任务到达“协助处理节点”
        task = taskService.createTaskQuery().taskName("协助处理").singleResult();    #2-S
        assertNotNull(task);
        taskService.complete(task.getId());    #2-E
        // 任务流转至“审核文件”节点
        task = taskService.createTaskQuery().taskName("审核文件").singleResult();    #3
        assertNotNull(task);
    }
}
```

代码清单11-11中#1处的3行代码在这里第一次接触到，其中#1-S处的ExecutionQuery查询对象是专门为查询流程执行对象（Execution，与ProcessInstance是多对一的关系）设计的，这里使用ExecutionQuery对象查询的原因是引擎遇到有附加消息边界事件的任务时会创建一个执行对象来监听消息事件的触发动作。图11-13的流程启动之后紧接着就会创建名称为“审核文件”的用户任务（Task对象），在创建过程中检测到存在附加的消息边界事件就会创建一个执行实例，同时会和消息启动事件一样在引擎中注册一个消息。

文件描述总归是抽象的，而观察数据记录（见图11-15）有助于理解这一过程。表ACT_RU_EVENT_SUBSCR中的记录的消息名称和图11-13中的消息边界事件的名称匹配。注意，该记录的“EXECUTION_ID_”和“PROC_INST_ID_”字段的值不为空，这与11.1.2节中的消息启动事件不同。这两个字段的值也是不同的，观察表ACT_RU_EXECUTION的记录可以说明这一点，ID_为70的记录是由ID_为67的记录创建的（通过PARENT_ID_字段可以证明）。另外还需注意ACT_RU_EXECUTION的两条记录的“ACT_ID_（表示一个执行对象的当前活动）”和“IS_ACTIVE_（是否处于活动状态）”字段值不同，也就是说主流程挂起，监听消息的执行对象处于活动状态。



ACT_RU_EVENT_SUBSCR

ID_	REV_	EVENT_TYPE_	EVENT_NAME_	EXECUTION_ID_	PROC_INST_ID_	ACTIVITY_ID_
71	1	message	协助处理	70	67	boundarymessage1



ACT_RU_EXECUTION

ID_	REV_	PROC_INST_ID_	BUSINESS_KEY_	PARENT_ID_	PRGC_DEF_ID_	SUPER_EXEC_	ACT_ID_	IS_ACTIVE_
67	1	67			messageBoundaryEvent:1:66			false
70	1	67		67	messageBoundaryEvent:1:66		usertask1	true

图11-15 启动了图11-13的流程之后数据记录

根据前面假设的业务流程现在应该把任务流转到“协助处理”节点，代码清单11-11的1-E处通过调用API方法触发了名称为“MSG_协助处理”的消息事件（此时取消了已注册的消息事件），消息边界事件的输出流指向了“协助处理”用户任务，所以在#2处查询到该任务并完成任务办理，最后在#3处验证了任务回归到“审核文件”节点。

图11-16用流程图的形式展示了该流程的执行过程，并说明了每个步骤做的任务。

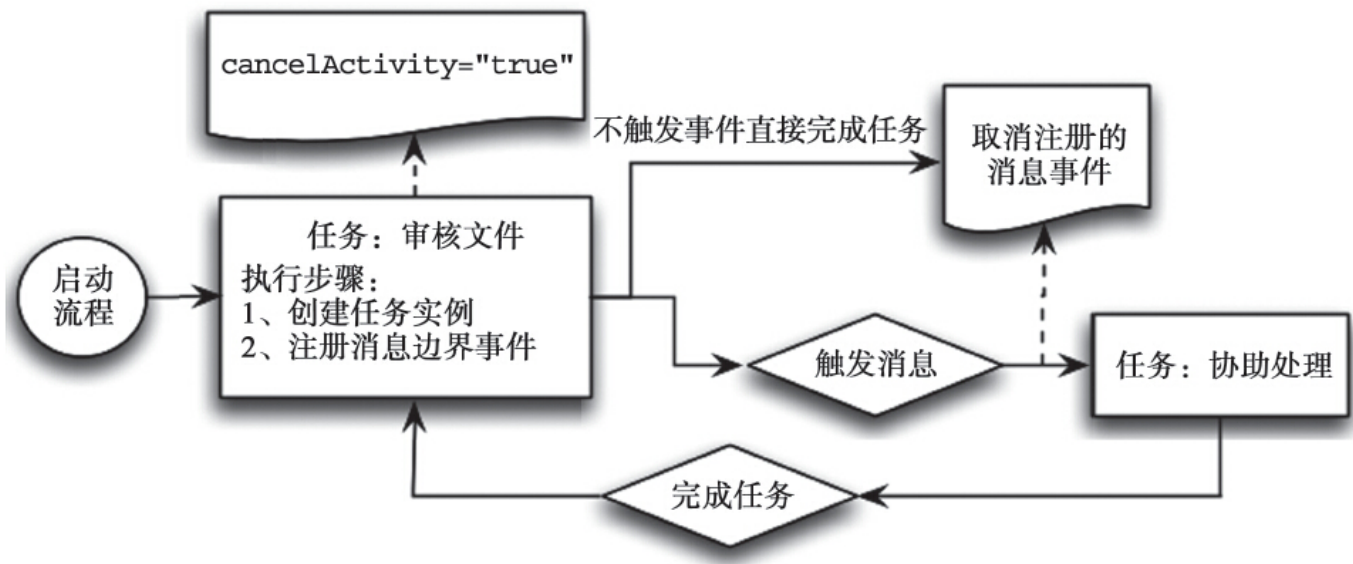


图11-16 包含消息边界事件的流程执行过程

注意 如果一个任务有附加的消息边界事件，当任务执行完成且边界事件没有被触发，那么已经注册的事件将在任务完成时被删除。

2. 多次触发

多次触发是相对于单次触发来说的，即设置“cancelActivity”属性为false，这样在触发了消息边界事件之后继续保留已注册的事件，可以连续多次触发，直到完成任务“审核文件”（任务完成时取消注册的消息事件）。

为了介绍多次触发，特意在messageBoundaryEvent.bpmn的基础上更改cancelActivity的值为“false”，并且另存了一个文件messageBoundaryEventNoCancelActivity.bpmn。

图11-17模拟的是多次触发事件的执行过程，与图11-16相比稍微有些区别，移除了“触发（多次）消息”后面的虚线，也就是说引擎不会在触发了消息之后取消已注册的消息事件。

代码清单11-12是根据图11-17的执行过程进行模拟，通过数据来分析多次触发后流程数据的变化。

代码清单11-12 多次触发消息边界事件单元测试

```

@Deployment(resources = "chapter11/boundaryEvent/messageBoundaryEventNoCancelActivity.bpmn")
public void testTriggerManyTimes() throws Exception {
    ProcessInstance processInstance = runtimeService.startProcessInstanceByKey("messageBoundaryEventNoCancelActivity");
    // 审核文件任务
    Task task = taskService.createTaskQuery().taskName("审核文件").singleResult();
    ExecutionQuery executionQuery = runtimeService.createExecutionQuery()
        .messageEventSubscriptionName("MSG_协助处理");
    Execution execution = executionQuery.singleResult();
    // 两次触发消息事件
    runtimeService.messageEventReceived("MSG_协助处理", execution.getId());
    runtimeService.messageEventReceived("MSG_协助处理", execution.getId());
    // 两个名称为“协助处理”的任务实例
    assertEquals(2, taskService.createTaskQuery().taskName("协助处理").count());
    // 完成两个协助处理任务
    for (Task task2 : list) {
        taskService.complete(task2.getId());
    }
    // 任务回归到审核文件，此时有3个注册了消息名称为“MSG_协助处理”的事件
    assertEquals(3, executionQuery.count());
    List<Execution> list3 = executionQuery.list();
    for (Execution execution2 : list3) {
        task = taskService.createTaskQuery().executionId(execution2.getId()).singleResult();
        taskService.complete(task.getId());
    }
    // 三个任务都完成了，此时整个流程实例结束
    assertEquals(1, historyService.createHistoricProcessInstanceQuery().finished().count());
}
  
```

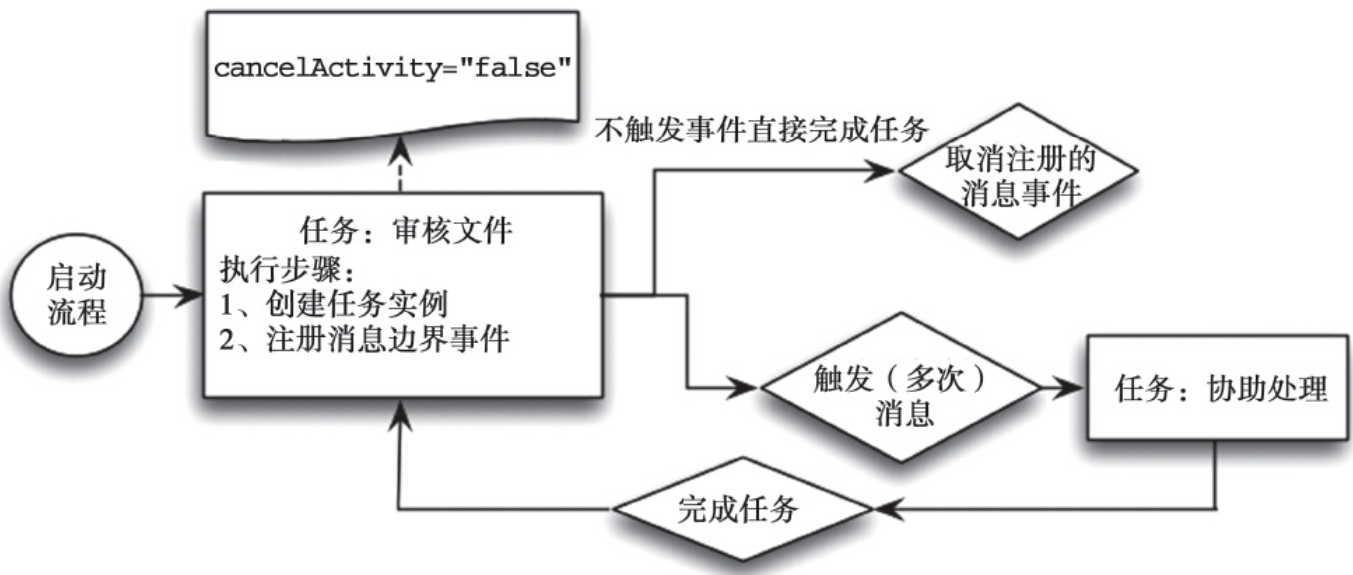


图11-17 多次触发消息边界事件执行过程

有了单次触发的基础，现在来理解代码清单11-12就比较容易了，根据图11-17的过程进行模拟，读者可以用JUnit运行该单元测试，观察输出结果（一些输出语句由于篇幅原因没有列出来）。

11.3.3 信号边界事件

信号事件与消息事件的事件处理机制是类似的。图11-18把图11-13的消息边界事件更改为信号边界事件。

从图11-18中可以看出，添加、编辑、设置信号的方式与消息事件类似，所以不再做过多的介绍，读者可以参考图11-13和图11-14。

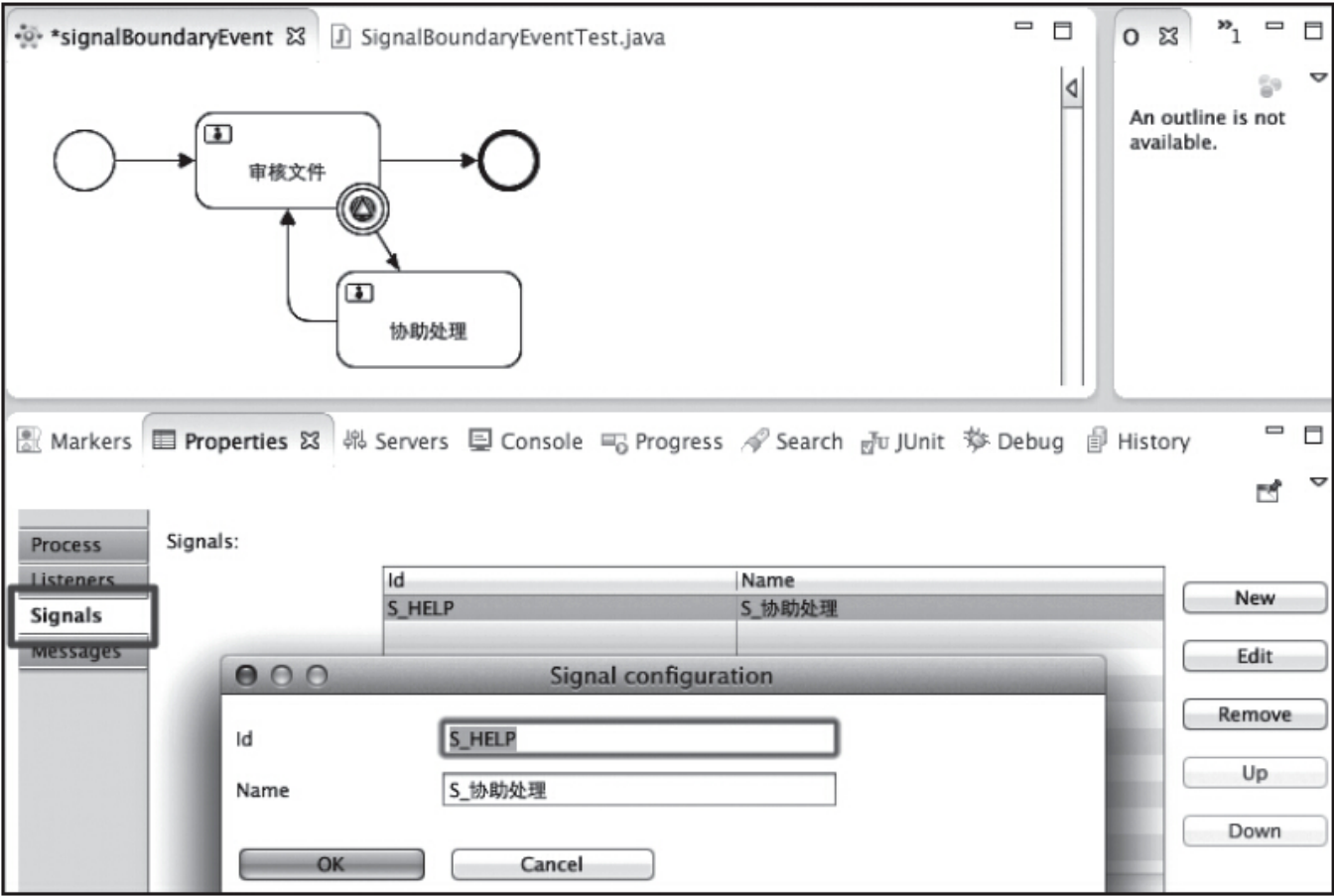


图11-18 包含信号边界事件的文档审批流程图

图11-18生成的XML代码与图11-13生成的XML代码类似，只不过把message换成了signal。下面列出了一些关键代码，可以与消息类型的边界事件进行对比。

```
<signal id="S_HELP" name="S_协助处理"></signal>
<process id="SignalBoundaryEvent" name="Signal Boundary Event" isExecutable="true">
  <boundaryEvent id="boundarysignal1" name="Signal" attachedToRef="usertask1" cancelActivity="true">
    <signalEventDefinition signalRef="S_HELP"/>
  </boundaryEvent>
</process>
```

由于信号边界事件与消息边界事件的处理机制类似，所以本节不在用大篇幅内容讲解，读者可以参考本章源码的SignalBoundaryEventTest.java单元测试类，该测试类中包含了属性cancelActivity的值为true和false两种情况。

特别说明 Java Service任务不支持定时边界事件，原因是Java Service任务可能会因运行时间过长或者Java类执行失败后事务回滚导致数据状态不一致。如果想要为边界事件添加一个定时任务，可以在实现类中利用其他的定时作业框架实现或者把任务发送到消息服务。

11.4 中间事件

根据11.3节中对边界事件的理解我们知道，边界事件是“捕获型”的，它可以监听其依附的活动上的某一种类型的事件。本节将要介绍的中间事件有抛出型和捕获型两种。之所以称之为中间事件是因为这类事件总是在两个输出流中间存在。图11-19展示了包含一个信号边界事件与信号中间捕获事件的流程。

在图11-19 (catchMultipleSignals.bpmn) 中有两个捕获类型的信号事件，左侧的信号中间捕获事件监听的信号名称为“abort”，用户任务“附加边界用户任务”附加的信号边界事件捕获的信号名称为“alert”。从该流程图可以看出在启动流程之后会向引擎注册两个信号事件（与注册消息事件类似），然后流程处于等待状态，在通过调用API或者主动在流程中抛出一个信号事件后该流程对应的信号事件将被触发并执行后续的输出流。

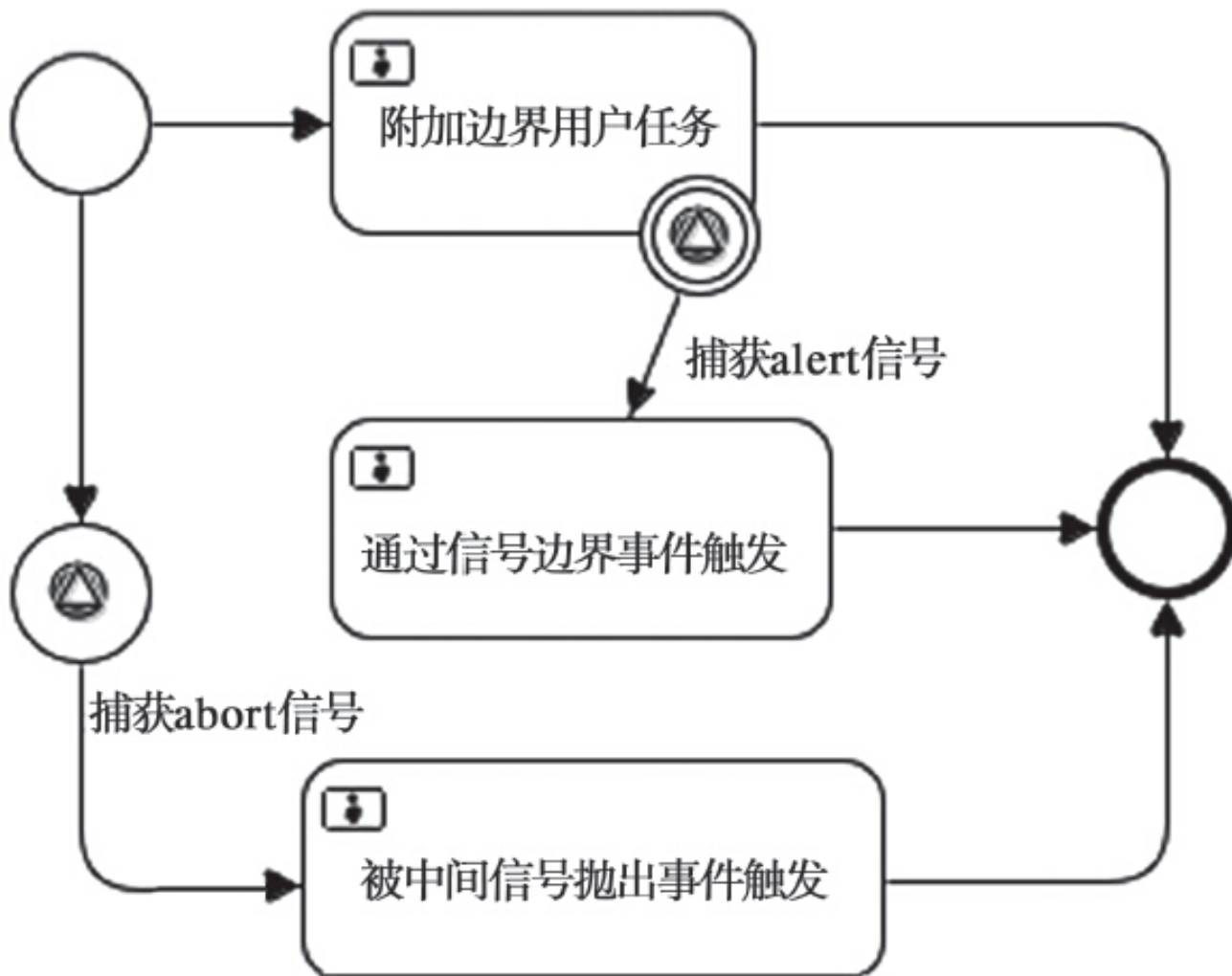


图11-19 包含信号中间捕获事件的流程图

为了了解信号事件的抛出与捕获，利用图11-20 (throwAbortSignal.bpmn) 和图11-21 (throwAlertSignal.bpmn) 中的两个简单流程模拟主动抛出信号事件。

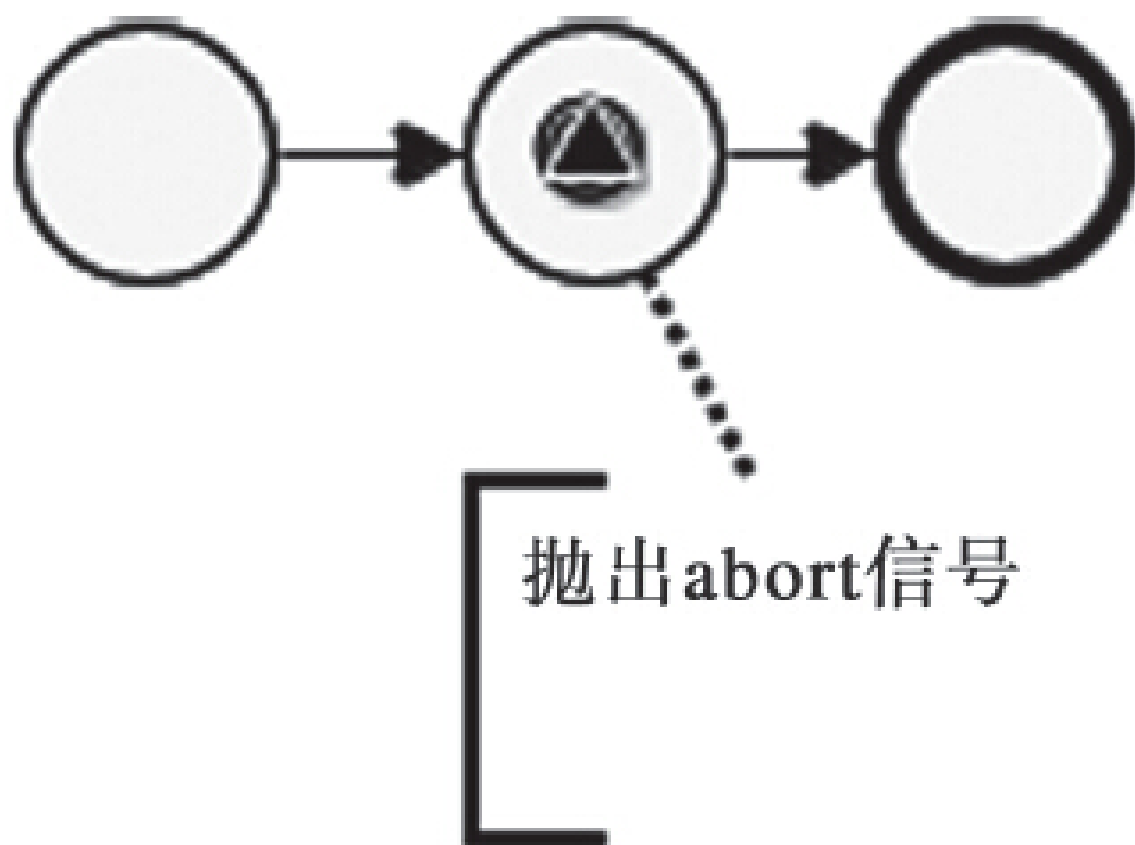


图11-20 抛出abort信号

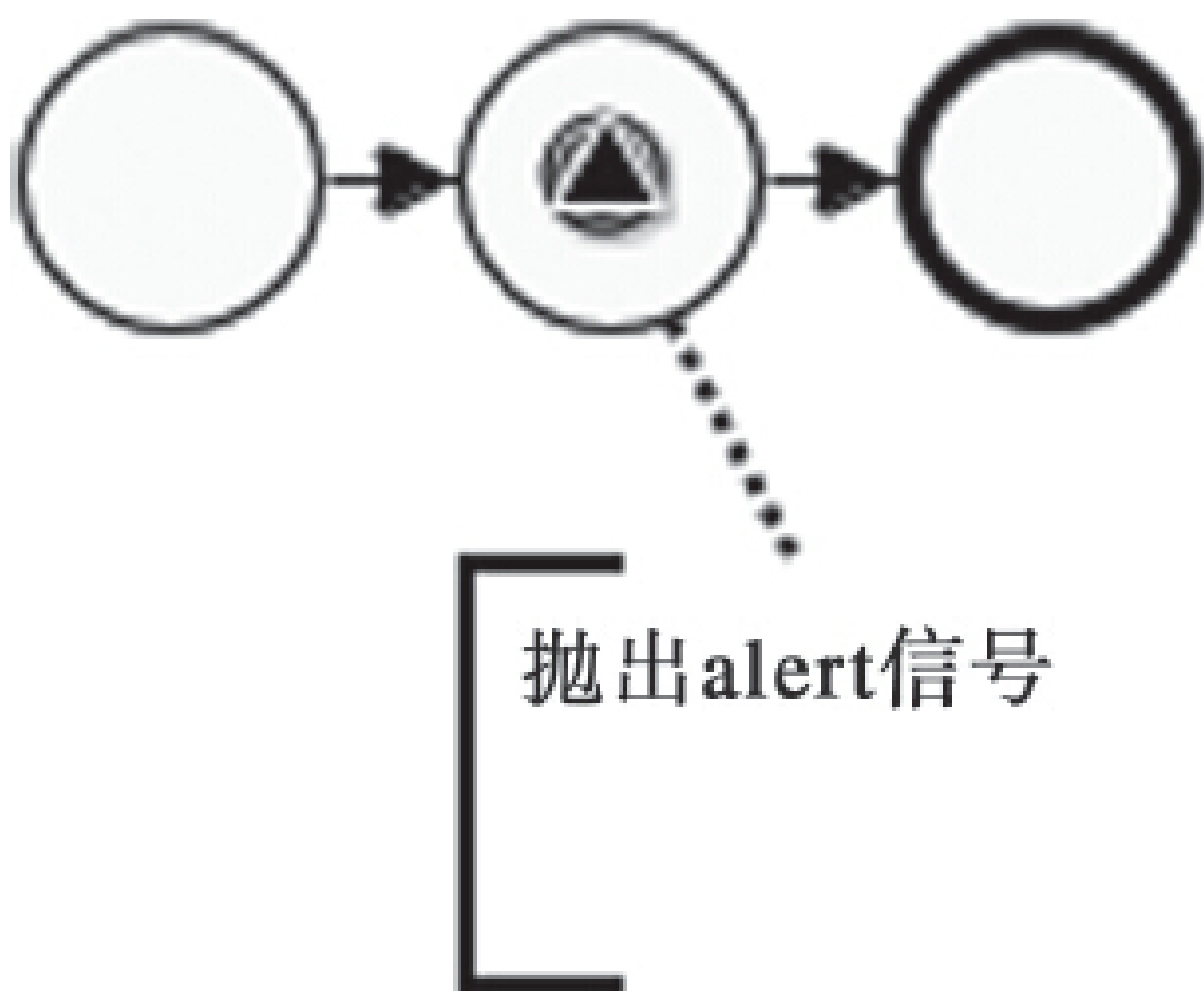


图11-20中的信号中间抛出事件对应的XML如下：

```
<intermediateThrowEvent id="signalintermediatethrowevent1"
    name="SignalThrowEvent">
    <signalEventDefinition signalRef="abort"/>
</intermediateThrowEvent>
```

图11-21中的信号中间抛出事件对应的XML如下：

```
<intermediateThrowEvent id="signalintermediatethrowevent1"
    name="SignalThrowEvent">
    <signalEventDefinition signalRef="alert"/>
</intermediateThrowEvent>
```

对应地，图11-19中的信号中间抛出事件与信号边界事件的XML如下：

```
<signal id="alert" name="alert"></signal>
<signal id="abort" name="abort"></signal>
<!-- 信号中间捕获事件 -->
<intermediateCatchEvent id="signalintermediatecatchevent1" name="SignalCatchEvent">
    <signalEventDefinition signalRef="abort"/>
</intermediateCatchEvent>
<!-- 信号边界事件 -->
<boundaryEvent id="boundarysignal1" name="Signal"
    attachedToRef="usertask1" cancelActivity="true">
    <signalEventDefinition signalRef="alert" />
</boundaryEvent>
```

为了将这三个流程联合起来了解抛出型与捕获型事件如何工作，我们通过一个单元测试来模拟流程的执行过程，见代码清单11-13。

代码清单11-13 信号中间（抛出、捕获）事件与边界事件测试

```
public class SignalIntermediateEventTest extends PluggableActivitiTestCase {
    @Deployment(resources = { "chapter11/intermediateEvent/signal/catchMultipleSignals.bpmn",
        "chapter11/intermediateEvent/signal/throwAlertSignal.bpmn",
        "chapter11/intermediateEvent/signal/throwAbortSignal.bpmn" })
    public void testSignalIntermediateEvent() throws Exception {
        runtimeService.startProcessInstanceByKey("catchSignal"); // 启动图11-19所示的流程
        // 注册了两个信号事件：alert、abort
        assertEquals(2, createEventSubscriptionQuery().count());
        assertEquals(1, runtimeService.createProcessInstanceQuery().count());
        runtimeService.startProcessInstanceByKey("throwAbort"); #1
        // 启动了图11-20所示的流程后抛出了abort信号，同时移除了注册的信号事件
        assertEquals(1, createEventSubscriptionQuery().count());
        assertEquals(1, runtimeService.createProcessInstanceQuery().count());
        Task taskAfterAbort = taskService.createTaskQuery()
            .taskName("被信号中间抛出事件触发").singleResult();
        // 图11-19中的中间信号捕获事件捕获到abort信号后创建了用户任务
        assertNotNull(taskAfterAbort);
        taskService.complete(taskAfterAbort.getId());
        runtimeService.startProcessInstanceByKey("throwAlert"); #2
        // 启动了图11-21的流程后抛出了alert信号，触发了“附加边界用户任务”上的
        // 信号边界事件，该任务随即被取消并创建任务“通过信号边界事件触发”
        Task taskAfterAlert = taskService.createTaskQuery()
            .taskName("通过信号边界事件触发").singleResult();
        assertNotNull(taskAfterAlert);
        taskService.complete(taskAfterAlert.getId());
        // 注册的事件都被触发，此时再次统计结果为0
        assertEquals(0, createEventSubscriptionQuery().count());
        assertEquals(0, runtimeService.createProcessInstanceQuery().count());
    }
}
```

通过代码清单11-13的执行过程分析了解了如何抛出、捕获信号事件。除了可以用信号中间抛出事件触发信号捕获事件之外，还可以用手动方式调用引擎API触发。代码清单11-13的#1与#2处可以用下面的两行代码替代，读者可以自行实践。

```
runtimeService.signalEventReceived("alert");
runtimeService.signalEventReceived("abort");
```

11.5 本章小结

事件是Activiti中比较重要的一块内容，本章先后介绍了启动事件、结束事件、边界事件、中间事件这4类事件。每一大类又根据不同功能进行了进一步划分，读者可以结合图标、测试用例来理解，根据业务需求的不同选择不同的事件类型以及触发事件的方式。

第12章 用户任务与附件

在一般流程中大部分活动都是任务处理，例如用户任务、Java Service任务、WebService任务、业务规则任务或者邮件任务等。在这几类任务中，除了用户任务需要人参与处理外其他任务都由引擎自动处理。

本章以用户任务为主线，介绍和用户任务有关的操作、原理，包括更改用户任务的属性，添加或者删除任务的候选人（组）、参与者，如何不通过流程创建任务，如何实现任务委派，以及和任务相关附件。

图12-1展示的是Activiti Explorer的任务办理界面，用箭头和文字对不同的区域进行了说明。本章的内容将围绕图12-1的每一个区域的功能点进行——讲解。



图12-1 Activiti Explorer任务办理界面

12.1 用户任务

首先要对现有的任务列表（如图12-2所示）进行改进，目前的问题是在没有签收任务时不能查看任务表单，需要改进的内容有下面两项：

- 在不签收的情况下可以查看表单。
- 查看（依赖第一项改进）表单时可以签收任务。

任务ID	任务名称	流程实例ID	流程定义ID	任务创建时间	办理人	操作
22	领导审批	11	purchase-subprocess:1:10	Thu Mar 14 19:52:11 CST 2013		✓ 签收

图12-2 待办任务列表

12.1.1 改进任务列表

首先改进的是待办任务列表，在不签收任务的情况下可以查看任务表单，单击图12-3中添加的“查看”按钮可以打开图12-4的任务表单界面。图12-3的“查看”按钮和图12-4的“签收”按钮通过任务的办理人（assignee）属性是否为空作为条件控制按钮是否显示。

任务ID	任务名称	流程实例ID	流程定义ID	任务创建时间	办理人	操作
22	领导审批	11	purchase-subprocess:1:10	Thu Mar 14 19:52:11 CST 2013		👁 查看 ✓ 签收

图12-3 改进后的任务列表

Activiti Explorer 首页 请假（普通表单） 任务列表 管理

任务办理—[领导审批]，流程定义ID: [purchase-subprocess:1:10]

申请人: kermi

到货期限: 2013-03-01

物品清单: 2432

总金额: 234.0

是否同意: 同意

返回列表

签收

添加意见

意见列表

1. dsfwfw kermi（领导审批） 3/14/2013 8:06:36 PM

+ 保存意见

根据任务是否拥有办理人
决定显示“签收”
还是“完成任务”按钮

图12-4 任务表单界面

图12-4任务表单中的签收按钮控制显示与否的逻辑如下（task-form.jsp）：

```
<c:if test="${not empty task.assigned}">
  <button type="submit">完成任务</button>
</c:if>
<c:if test="${empty task.assigned}">
  <a href="${ctx }/chapter6/task/claim/${task.id}
    ?nextDo=handle">签收</a>
</c:if>
```

“签收”按钮传递了一个参数nextDo，表示签收之后的动作，前面章节的例子中签收按钮只在任务列表显示，单击了之后签收任务便回到任务列表。如果单击任务表单中的“签收”按钮，不仅需要签收任务还需要重新打开任务表单，以便显示“完成任务”按钮，单击此按钮可以提交任务。

后台任务签收控制器（TaskController.java#claim）也需要修改以支持nextDo参数的不同取值。下面的代码展示了针对nextDo参数的值为“handle”时的处理：

```
if (StringUtils.equals(nextDo, "handle")) {
  return "redirect:/chapter6/task/getform/" + taskId;
} else {
  return TASK_LIST; // 在默认情况下签收之后回到任务列表
}
```

稍加改造后的任务列表与任务表单就可以满足我们预期的两个功能，当单击了图12-4的“签收”按钮之后，“完成任务”按钮就显示出来了，如图12-5所示。



图12-5 在任务表单中单击“签收”按钮后显示“完成任务”按钮

在图12-1的顶部除了显示任务名称之外还有下面几个属性：

- 任务到期日（No due date）。设置了这个属性之后可以提前一段时间向用户发送通知，例如通过邮件、短信等方式。
- 任务优先级（Medium Priority）。任务的优先级可以从1~100，一般约定0表示低，50表示中，100表示高；当然分级的粒度还可以更细，例如，对于引擎，建议像下面这样分级：
 - [0http://www.hzcourse.com/resource/readBook?path=/openresources/teach_ebook/uncompressed/15030/OEBPS/Text/..19]表示最低级
 - [20http://www.hzcourse.com/resource/readBook?path=/openresources/teach_ebook/uncompressed/15030/OEBPS/Text/..39]表示低级
 - [40http://www.hzcourse.com/resource/readBook?path=/openresources/teach_ebook/uncompressed/15030/OEBPS/Text/..59]表示一般
 - [60http://www.hzcourse.com/resource/readBook?path=/openresources/teach_ebook/uncompressed/15030/OEBPS/Text/..79]表示高级
 - [80http://www.hzcourse.com/resource/readBook?path=/openresources/teach_ebook/uncompressed/15030/OEBPS/Text/..100]表示最高级
- 任务创建时间。

添加了几个属性后效果如图12-6所示，从图中可以看出在任务名称下方添加了任务到期日（dueDate）、优先级（priority）、创建日期（createTime）。查看引擎的API文档可以看到org.activiti.engine.Task的getter方法，以此类推可以在任务表单中显示更多的属性。

任务办理—[领导审批]，流程定义ID：[purchase-subprocess:1:10]

到期日：无到期日

优先级：中

创建日期：2013-03-14 07:52:11

申请人：

kermi

图12-6 在任务表单中添加了任务属性

下面的代码列出了显示任务属性的HTML代码，其中利用<c:if/> 标签进行了条件判断：

```
<span><i class="icon-calendar"></i>到期日：
  <c:if test="{empty task.dueDate}">无到期日</c:if>
  <c:if test="{not empty task.dueDate}">
    ${task.dueDate}</c:if>
</span>
<span><i class="icon-flag"></i>优先级：
  <c:if test="{empty task.priority}">无到期日</c:if>
  <c:if test="{not empty task.priority}">
    <c:if test="{task.priority = 0}">低</c:if>
    <c:if test="{task.priority = 50}">中</c:if>
    <c:if test="{task.priority = 100}">高</c:if>
  </c:if>
</span>
<span><i class="icon-calendar"></i>创建日期：
  <fmt:formatDate value="{task.createTime}"
    pattern="yyyy-MM-dd hh:mm:ss"/>
</span>
```

细心的读者可能会发现，单击图12-1的任务属性后可以直接更改任务属性的值，如图12-7所示。在任务属性的值更改了之后可以利用Ajax () 向后台发送请求，从而调用引擎API更改任务属性。

Handle vacation request

3/16/13 Medium Priority Created moments ago

Claim

kermi would like to

Part of process: 'Vacation request'

People

No owner

Transfer

March 2013

SUN	MON	TUE	WED	THU	FRI	SAT
24	25	26	27	28	1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31	1	2	3	4	5	6

图12-7 单击任务属性可以更改任务属性的值

为了学习如何调用引擎的API更改任务属性，对图12-6中的任务属性进行再次改造，利用html组件的change（在组件的值更改后可以回调一个指定的函数）事件监听实现异步发送更改属性请求。

经过改造后再次单击任务属性时就会看到如图12-8所示的任务属性编辑功能，在选择了日期后，失去输入框的焦点即可自动发送异步的POST请求，后台响应成功后将日期更改为选择的日期。



图12-8 单击任务属性可以进行更改

由于动态任务属性使用的jQuery不在本书讨论范围之内，因此这里对实现细节不做讨论，读者可以自行查看源码了解实现过程（chapter12-task-attachment中的task-form.js文件）。处理请求的Java代码才是本节的重点内容，根据请求参数更改任务属性的代码如代码清单12-1所示。

代码清单12-1 更改任务属性的控制器方法

```
@RequestMapping("task/property/{taskId}")
@ResponseBody
public String changeTaskProperty(@PathVariable("taskId") String taskId,
    @RequestParam("propertyName") String propertyName,
    @RequestParam("value") String value) {
    // 根据任务ID查询任务对象
    Task task = taskService.createTaskQuery().taskId(taskId).singleResult();
    if (StringUtils.equals(propertyName, "dueDate")) { // 更改到期日
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
        Date parse = sdf.parse(value);
        task.setDueDate(parse); // 设置到期日
        taskService.saveTask(task); // 更新任务属性
    } else if (StringUtils.equals(propertyName, "priority")) { // 更改任务优先级
        task.setPriority(Integer.parseInt(value)); // 设置优先级属性
        taskService.saveTask(task);
    }
    return "success";
}
```

12.1.3 任务相关人员

在图12-1中的“People”一栏中可以看到任务的拥有人（owner）、任务办理人（assignee），以及其他类别的人员。下面分别说明用户任务相关的几类人员的作用。

- **拥有人**：在流程处理过程中产生的用户任务，其拥有人属性为空，可以通过调用API方式更改用户任务的拥有人
- **办理人**：从前面章节介绍的任务办理内容中已经有所了解，如果将一个任务直接分配给某一个用户，则这个用户就是任务的办理人；如果用户包含在候选人列表中或属于候选组中，并且签收（claim）了该任务，此时该用户就是该任务的办理人。除了这两种方式还可以像更改任务到期日一样更改任务的办理人属性
- **参与人**：可以为用户任务添加不同类型（可以是任意类型）的人员参与该任务办理，例如添加一个用户参与该任务的贡献者（根据任务内容给出办理意见，可以将意见以批注的方式分享给参与人）

本小节把这三类人员分别添加到任务表单中，以了解如何更改、添加人员及其用途。

设置任务的拥有人、办理人的方式与12.1.2节中设置任务到期日的方式类似，因为拥有人、办理人都是任务的一个属性，更新属性值后保存任务即可；参与人不能简单地以设置任务属性的方式设置，因为参与人是一个列表而不是一个属性（即数据表现形式是一个列表），为任务添加参与人实际上是向表（ACT_RU_IDENTITYLINK）中插入一条数据。

使用相同的方式设置任务拥有人、办理人，在任务表单JavaScript与控制器中添加类似的处理代码即可很容易地完成这两个属性的设置功能，图12-9展示了最终效果。

控制器在代码清单12-1的基础上添加对于拥有人与办理人的支持，代码片段如下，JavaScript也做了对应的改进与支持。

```
if (StringUtils.equals(propertyName, "owner")) {
    // 更改拥有人
    task.setOwner(value);
    taskService.saveTask(task);
} else if (StringUtils.equals(propertyName, "assignee")) {
    // 更改办理人
    task.setAssignee(value);
    taskService.saveTask(task);
}
```

任务办理—[领导审批]， 流程定义ID: [purchase-call

到期日： 无到期日

优先级： 高

创建日期： 2014-11-20 12:09:16

任务人员

拥有人：

✓ Amy Zhang

Andy Zhao

Bill Zheng

Eric Li

Henry Yan

Jenny Luo

参与人

候选[人|组]

-- 候选组

✕ 删除

办理人： kermi

任务委派： 无

图12-9 添加了任务人员设置功能

对于参与人，用现实的场景可能更容易理解，例如一个项目聘请了业务顾问，当讨论某一个需求时，可以邀请顾问参与进来，顾问只是一种角色或者职责，因此可以根据具体的制度（公司岗位的划分）预设几种类别，通过添加几个预设的类别作为任务的候选类型，如贡献人、项目经理、总经理、业务顾问、技术顾问、执行人。

图12-10展示了在Activiti Explorer中添加参与人的界面，被邀请的用户将有权限查看该任务，可以对任务内容做出响应（通过添加意见的方式互动）。图12-11展示的是改造后的任务人员区域，在“参与人”部分可以看到已经被邀请的三个参与人，单击后面的“删除”链接可以删除一个参与人。添加参与人后单击“邀请”按钮弹出如图12-12所示的对话框。

People

No owner

Transfer

Kermit The Frog

Assignee

Reassign

+

Involve people

Search people ...

Select myself

Kermit The Frog

Contributor

✕

图12-10 在Activiti Explorer中添加参与人

任务人员

+ 邀请

拥有人： henry

办理人： kermi

参与人

• amy -- 贡献人

✕ 删除

• henry -- 总经理

✕ 删除

• jenny -- 技术顾问

✕ 删除

图12-11 改造后的任务人员区域

邀请参与人

单击可添加一行

	人员	职能	操作
1	Amy Zhang	贡献人	
2	Bill Zheng	总经理	

关闭

确定邀请

图12-12 邀请参与人对话框

单击了图12-12的“确定邀请”按钮后通过Ajax方式发送请求至后台控制器，成功后刷新页面即可看到图12-11所示的参与人列表。

由于参与人页面操作的相关JavaScript代码比较多且不属于本书讨论范围，故不作过多的解释，重点是引擎提供的API如何使用。

代码清单12-2列出了添加参与人的相关代码，方法addParticipants()接收数组形式的用户ID和类型ID，分别对应图12-12中的两个下拉框选择的值，然后循环多次调用引擎的TaskService#addUserIdentityLink()添加参与人。对应地，也可以调用deleteUserIdentityLink()方法删除一个参与人。

代码清单12-2 为任务添加参与人的控制器代码

```
@RequestMapping("task/participant/add/{taskId}")
@ResponseBody
public String addParticipants(@PathVariable("taskId") String taskId,
    @RequestParam("userId[]") String[] userIds, @RequestParam("type[]")
    String[] types) {
    String currentUser = UserUtil.getUserFromSession(request.getSession()).getId();
    identityService.setAuthenticatedUserId(currentUser); // 记录当前操作人
    for (int i = 0; i < userIds.length; i++) {
        taskService.addUserIdentityLink(taskId, userIds[i], types[i]);
    }
    return "success";
}
```

读取参与人列表也比较简单，在读取任务表单的控制器方法（代码清单6-26）中添加下面的代码，读取到一个List集合，将该集合添加到ModelAndView对象中，如此即可在JSP页面中循环输出参与人。

```
List<IdentityLink> identityLinksForTask =
    taskService.getIdentityLinksForTask(taskId);
mav.addObject("links", identityLinksForTask);
```

通过调用IdentityLink对象的getter方法可以获取和任务、参与人有关的数据，其API如图12-13所示。在图12-11中显示参与人的代码可以参见task-form.jsp文件。

Method and Description

getGroupId()

If the identity link involves a group, then this will be a non-null id of a group.

getTaskId()

The id of the task associated with this identity link.

getType()

Returns the type of link.

getUserId()

If the identity link involves a user, then this will be a non-null id of a user.

图12-13 IdentityLink对象的API

单击图12-11中的“删除”链接后即可删除一个参与人，也就是从ACT_RU_IDENTITYLINK表中删除一条数据，代码清单12-3列出了删除参与人的代码。

代码清单12-3 删除参与人的控制器代码

```
@RequestMapping("task/participant/delete/{taskId}")
@ResponseBody
public String deleteParticipant(@PathVariable("taskId") String taskId,
    @RequestParam(value = "userId") String userId, // 用户ID
    @RequestParam("type") String type) { // 参与人类型
    taskService.deleteUserIdentityLink(taskId, userId, type);
    return "success";
}
```

12.1.4 反签收任务

反签收是相对于签收动作来说的，顾名思义，可以把已经签收的任务的办理人置空，这样任务又还原到未签收状态，此功能对于日常使用中误签收的情况非常有用。

反签收任务的API与签收任务的相同，唯一不同的是在签收任务时需要传递任务ID和用户ID，而反签收只需要传递任务ID，用户ID参数以“null”代替即可。下面的代码展示了两者的不同：

```
taskService.claim(taskId, userId); // 签收，userId不能为空
taskService.claim(taskId, null); // 反签收，第二个参数为空
```

把反签收功能加入到任务列表中，根据任务的办理人是否为空判断是否显示反签收按钮，图12-14展示的是已签收的任务，图12-15展示的是单击了“反签收”按钮后的任务列表。

任务已签收						
任务ID	任务名称	流程实例ID	流程定义ID	任务创建时间	办理人	操作
22	领导审批	11	purchase-subprocess:1:10	Thu Mar 14 19:52:11 CST 2013	kermit	<button>办理</button> <button>反签收</button>

图12-14 签收任务后的任务列表

任务已反签收						
任务ID	任务名称	流程实例ID	流程定义ID	任务创建时间	办理人	操作
22	领导审批	11	purchase-subprocess:1:10	Thu Mar 14 19:52:11 CST 2013		<button>查看</button> <button>签收</button>

图12-15 单击“反签收”按钮后的任务列表

在执行反签收时有一点要特别注意，如果用户任务没有相关候选人与候选组，执行反签收动作会导致任务无人认领（可以理解为游离状态），所以在反签收时要判断该任务是否满足反签收条件。可以通过查询任务是否有候选人或者候选组的方式验证，如代码清单12-4所示。

代码清单12-4 反签收动作附带条件检验

```
// 反签收条件过滤
List<IdentityLink> links = taskService.getIdentityLinksForTask(taskId);
for (IdentityLink identityLink : links) {
    // 如果一个任务有相关的候选人、候选组就可以反签收
    if (StringUtils.equals(IdentityLinkType.CANDIDATE, identityLink.getType())) {
        taskService.claim(taskId, null);
        break;
    }
}
```

代码清单12-4首先查询和任务相关的人员，循环判断IdentityLink对象的候选类型是否为“candidate”，如果不是“candidate”，则不执行反签收，在一定程度上可以解决任务反签收后导致无人认领的问题。

12.1.5 候选人与候选组

在12.1.3节中了解了通过TaskService#addUserIdentityLink(taskId, userId, identityLink-Type)方法可以添加一个参与者，其中第三个参数identityLinkType支持多种参与者关系类型。引擎对于每一种参与者关系类型都有不同的处理方式，具体如下：

- assignee，任务的办理人，不会向表ACT_RU_IDENTITYLINK中插入数据，和Task#setAssignee()的功能一样（实际就是调用了这个方法后更新任务）。
- owner，任务的拥有人，与assignee一样，不插入数据仅仅调用Task#setOwner()后更新任务。
- candidate，表示任务的候选（人或组），根据表中的两个字段（USER_ID_、GROUP_ID_）是否为空来判断是候选人还是候选组。
- starter，Activiti 5.12版本新添加的类型，如果在启动流程时设置了认证用户，则会向表ACT_RU_IDENTITYLINK中插入一条数据表示流程启动人。
- participant，Activiti 5.12版本新添加的类型，在添加候选人、候选组或者非引擎支持的关系类型时引擎会以流程实例ID作为标志（仅设置PROC_INST_ID_字段值不设置TASK_ID_字段值）插入数据到表ACT_RU_IDENTITYLINK中，方便根据流程实例查询参与人数据。

 **注意** 候选人与参与人是区别的：候选人有权限签收任务，参与者一般没有权限签收任务，仅提供参考意见。

图12-16比图12-11多出了一个“添加候选[人|组]”按钮，单击此按钮弹出图12-17所示的“添加候选[人|组]”对话框，可以根据“候选类型”选择候选人或候选组。

图12-17中的“候选类型”有两类（user：候选人，group：候选组），在更改了候选类型后，在“人/组”显示对应的下拉框（通过JavaScript控制自动隐藏对方），单击按钮“确定添加”将收集数据发送给代码清单12-5中的控制器方法。

任务办理—[领导审批]，流程定义ID：

到期日：2013-03-29

优先级：高

创建日期

任务人员

+ 邀请参与者

+ 添加候选[人|组]

拥有者：amy

办理人：无

参与者

候选[人|组]

• amy -- 贡献人（参与者）

✕ 删除

• henry -- 总经理（参与者）

✕ 删除

• jenny -- 技术顾问（参与者）

✕ 删除

• deptLeader -- 候选组

✕ 删除

• amy -- 候选人

✕ 删除

图12-16 添加了候选[人|组]按钮及列表

添加候选[人|组]

+

候选类型

人/组

操作

1

候选人

Amy Zhang

2

候选组

总经理

关闭

确定添加

图12-17 添加候选[人|组]对话框

代码清单12-5 添加候选人、候选组的控制器代码

```
@RequestMapping("task/candidate/add/{taskId}")
@ResponseBody
public String addCandidates(@PathVariable("taskId") String taskId,
    @RequestParam("userOrGroupIds[]") String[] userOrGroupIds, // 用户ID或组ID
    @RequestParam("type[]") String[] types) { // 类型 (user, group)
    for (int i = 0; i < userOrGroupIds.length; i++) {
        String type = types[i];
        if (StringUtils.equals("user", type)) { // 不同的类型调用不同的API方法
            taskService.addCandidateUser(taskId, userOrGroupIds[i]);
        } else if (StringUtils.equals("group", type)) {
            taskService.addCandidateGroup(taskId, userOrGroupIds[i]);
        }
    }
    return "success";
}
```

这里需要特别说明，通过TaskService#addCandidateUser、TaskService#addCandidateGroup添加的候选类型均为“candidate”，所以在任务表中显示候选人与候选组时只可以一并处理，见如下代码：

```
<c:when test="${link.type == 'candidate'}">
    <c:set var="type"
        value="候选${not empty link.userId ? '人' : '组'}" />
</c:when>
```

可以动态添加候选人、候选组，对应地也可以删除已添加的候选人、候选组，代码清单12-3已经展示了如何删除一个参与人。添加参与人与添加参与组在引擎内部实现其实调用的是同一个API，唯一不同的是参与的类型，所以删除参与人与删除参与的方法组也类似。但是候选组与候选人又有所不同，因此需要根据不同的情况加以判断，然后再调用对应的删除方法。

在代码清单12-3的基础上添加了支持删除候选人、候选组的功能，判断的依据是当参数groupId不为空时调用删除候选组方法，其他的情况和删除参与人的方法相同，具体见代码清单12-6所示。

代码清单12-6 支持删除参与人、候选人、候选组

```
@RequestMapping("task/participant/delete/{taskId}")
@ResponseBody
public String deleteParticipant(@PathVariable("taskId") String taskId,
    @RequestParam(value = "userId", required = false) String userId,
    @RequestParam(value = "groupId", required = false) String groupId,
    @RequestParam("type") String type) {
    if (StringUtils.isNotBlank(groupId)) {
        taskService.deleteCandidateGroup(taskId, groupId); // 对于候选组特殊处理
    } else {
        taskService.deleteUserIdentityLink(taskId, userId, type);
    }
    return "success";
}
```

12.1.6 改进任务查询

利用前面几个小节介绍了如何动态添加参与人、候选人、候选组，其目的只有一个——让更多的人参与任务办理。现在将话题回归到任务列表，可能有的读者在添加了参与人之后就退出参与人登录系统，可惜任务列表中没有任务数据，但是在动态添加候选人与候选组后可以查询到相关任务，这是为什么呢？在12.1.3节中实现的添加参与人功能可以从一个列表中选择自定义的参与人类型（表示岗位），所以原本的任务查询功能仅能查询到已签收的或者在候选人、候选组范围内的任务。

代码清单6-10中查询任务列表为了兼顾已签收和未签收的任务，查询两次把结果合并成一个集合传递给显示层，这里为了解决不能查询参与人的问题要对代码清单6-10的代码做一些小改动。

把原来根据候选人查询任务的taskCandidateUser()改为taskInvolvedUser()即可，最终查询任务的代码如代码清单12-7所示。

代码清单12-7 最终查询任务


```
// 读取直接分配给当前人或者已经签收的任务
List<Task> doingTasks = taskService.createTaskQuery()
    .taskAssignee(user.getId()).list();
// 受邀任务
List<Task> involvedTasks = taskService.createTaskQuery()
    .taskInvolvedUser(user.getId()).list();
// 合并两种任务
List<Task> allTasks = new ArrayList<Task>();
allTasks.addAll(doinTasks);
allTasks.addAll(involvedTasks);
```

从图12-16中可以看出用户“henry”不是任务的办理人也不是候选人，更改了查询方式之后再次查看任务列表可以看到该任务，如图12-18所示。

Activiti Explorer						
首页 请假（普通表单） 任务列表 管理 Henry Yan/henry						
任务ID	任务名称	流程实例ID	流程定义ID	任务创建时间	办理人	操作
22	领导审批	11	purchase-subprocess:1:10	Thu Mar 14 19:52:11 CST 2013		查看 签收

图12-18 用户henry的任务列表

taskInvolvedUser()与taskCandidateUser()的区别就在于前者包含了参与者、候选人，以及属于某个候选组的人，而后者只在候选人、候选组范围内查询。

说明 用户任务相关的参与者一般不能签收任务，所以可以在签收时检查一下当前人是否在候选人、候选组范围之内，否则不执行签收动作。

12.2 子任务

在办理任务时可能会遇到需要把任务分解的情况，例如拆分成多个子任务交给不同的人办理，在Activiti中可以通过TaskService接口创建子任务或者根据父任务查询子任务。

创建子任务的方式相对来说比较简单，通过API创建一个任务并设置父任务ID，最后保存即可，当然可以同时设置该任务的其他属性（可以是Task对象支持的所有属性）。

图12-19展示了表单中的子任务区域，单击“添加”按钮创建一个子任务，在按钮的下方面列出了子任务列表，并且根据任务是否已结束决定是否显示“完成时间”文字。

任务办理—[领导审批]， 流程定义ID: [purchase-subprocess:1:10]

到期日: 2013-03-29

优先级: 高

创建日期: 2013-03-14 07:52:11

该任务无描述。

任务描述信息

任务人员

+ 邀请参与者

拥有人: amy

办理人: kermi

参与者

候选[人/组]

amy -- 贡献人 (参与者) ✕ 删除

henry -- 总经理 (参与者) ✕ 删除

jenny -- 技术顾问 (参与者) ✕ 删除

deptLeader -- 候选组 ✕ 删除

amy -- 候选人 ✕ 删除

子任务

+ 添加

协作审批 (完成时间: 2013-03-17 09:27:02)

子任务1111 ✕ 删除

子任务2222 ✕ 删除

已完成的子任务

未完成的子任务

图12-19 在任务表单中添加了子任务区域

单击图12-19中的“添加”按钮会弹出如图12-20所示的对话框，任务名称和任务描述与Task对象的属性对应，为了方便讲解这里仅定义了两个参数，其他的参数读者可以自行添加。单击“确定添加”按钮后提交表单，然后在后台控制器中根据请求参数创建任务并设置任务的父任务ID，实现代码见代码清单12-8。

添加子任务

任务名称:

子任务333

任务描述:

子任务描述信息

关闭

确定添加

图12-20 添加子任务对话框

代码清单12-8 添加子任务控制器

```
@RequestMapping("task/subtask/add/{taskId}")
public String addSubTask(@PathVariable("taskId") String parentId,HttpSession session,
    @RequestParam("taskId") String taskName,
    @RequestParam(value = "description", required = false) String description) {
    Task newTask = taskService.newTask(); // 创建一个新的任务
    newTask.setParentTaskId(parentId); // 设置父任务ID, 建立上下级关系
    String userId = UserUtil.getUserFromSession(session).getId();
    newTask.setOwner(userId); // 设置拥有人
    newTask.setAssignee(userId); // 设置办理人
    newTask.setName(taskName); // 设置任务名称
    newTask.setDescription(description); // 设置任务描述
    taskService.saveTask(newTask); // 保存任务
    return "redirect:/chapter6/task/getform/" + parentId;
}
```

刷新页面之后即可看到名称为“子任务333”的子任务，如图12-21所示。单击子任务列表的链接即可打开任务办理界面，如图12-22所示。当然可以继续在子任务中添加子任务，最终将形成一棵树状结构。

子任务

+ 添加

✓ 协助审批（完成时间：2013-03-17 09:27:02）

子任务1111

✕ 删除

子任务2222

✕ 删除

子任务333

✕ 删除

图12-21 添加的子任务

任务办理—[子任务333]

到期日: 无到期日

优先级: 中

创建日期: 2013-03-19 07:02:27

该任务属于: 领导审批

子任务描述信息

图12-22 子任务办理界面

第6章介绍了通过TaskService#createTaskQuery()创建TaskQuery对象后根据条件查询任务集合，也介绍了引擎为提高运行速度把流程数据状态划分为运行中与历史（也可称之为归档）两种类型，每

一种状态对应不同的数据表，在一个用户任务执行完成后将该任务从运行时任务表（ACT_RU_TASK）中删除，并设置历史任务（ACT_HI_TASKINST表）的结束时间，因为这两张表的主键ID是相同的，所以可以根据运行时任务ID查询历史任务，当然如果一个历史任务还没有结束，也可以查询运行时任务。

为了兼容运行时与历史类型的任务数据需要利用HistoryService接口查询历史任务实例，在代码清单6-26中追加代码清单12-9的代码用于查询子任务及上级任务。

代码清单12-9 查询子任务及上级任务

```
// 读取子任务
List<HistoricTaskInstance> subTasks = historyService.createHistoricTaskInstanceQuery()
    .taskParentTaskId(taskId).list();
mav.addObject("subTasks", subTasks);
// 读取上级任务
if (task != null && task.getParentTaskId() != null) {
    HistoricTaskInstance parentTask = historyService.createHistoricTaskInstanceQuery()
        .taskId(task.getParentTaskId()).singleResult();
    mav.addObject("parentTask", parentTask);
}
```

在任务表单页面循环输出ModelAndView对象中的“subTasks”即可得到如图12-21所示的子任务列表，根据单个任务的“结束时间”是否为空决定如何显示任务信息。图12-21中的第一条子任务已经处理完成，所以显示的图标与未完成的任务不同，并且显示了任务完成时间。

要办理任务，单击图12-21中的任务链接打开任务表单，但是需要区分该任务是否已结束，如果已经结束，需要使用历史任务查询功能并显示只读的任务界面，如果任务未结束且该任务的办理人与当前登录人匹配，可以办理任务。

代码清单12-10展示了历史任务控制器，可以根据任务ID查询历史任务、父任务以及子任务，最后返回只读的任务界面。

代码清单12-10 历史任务查询控制器

```
@RequestMapping(value = "task/archived/{taskId}")
public ModelAndView viewHistoryTask(
    @PathVariable("taskId") String taskId) throws Exception {
    String viewName = "chapter6/task-form-archived";
    ModelAndView mav = new ModelAndView(viewName);
    HistoricTaskInstance task = historyService.createHistoricTaskInstanceQuery()
        .taskId(taskId).singleResult();

    if (task.getParentTaskId() != null) {
        HistoricTaskInstance parentTask = historyService.createHistoricTaskInstanceQuery()
            .taskId(task.getParentTaskId()).singleResult();
        mav.addObject("parentTask", parentTask);
    }
    mav.addObject("task", task);
    // 读取子任务
    List<HistoricTaskInstance> subTasks = historyService.createHistoricTaskInstanceQuery()
        .taskParentTaskId(taskId).list();
    mav.addObject("subTasks", subTasks);
    return mav;
}
```

代码清单12-10中返回的视图“task-form-archived”与任务表单“task-form”视图类似，仅仅显示任务相关数据，不可以添加、更改。

第9章介绍了与任务相关的意见通过流程实例ID查询，但是添加的子任务没有设置流程实例ID，所以对于子任务而言就不能正常添加、读取意见列表了，为此需要对第9章中的代码做一些更改以支持子任务。

在第9章中根据流程实例ID查询Comment，而对于子任务没有流程实例的情况需要使用下面的代码查询Comment：

```
List<Comment> comments = taskService.getTaskComments(taskId);
```

12.3 手动任务

在Activiti Explorer的主界面中单击“New Task”按钮，弹出图12-23所示的对话框，与新增子任务的界面类似，通过这种方式创建的任务和流程无关，可以作为日常办公中的一个辅助功能。

New Task

Name *

Description

Duedate

Low Priority

Create

图12-23 Activiti Explorer的创建新任务对话框

如何创建一个新任务在介绍子任务时已经详细讲解了，所以本节只给出实现的效果，读者可自行参考本书配套资源中的代码实现。

首先在任务列表添加了一个“新任务”按钮，如图12-24所示，单击此按钮弹出如图12-25所示的新任务对话框，相比子任务的新增对话框多出了优先级、到期日属性，然后单击“创建”按钮提交表单，刷新页面显示如图12-26所示的任务表单。

任务列表		管理		Kermit Miao/kermit	
	父任务ID	任务创建时间	办理人	操作	+ 新任务

图12-24 在任务列表中添加了“新任务”按钮

新任务

任务名称:

去XXX客户签订合同

任务描述:

携带资料: 告知书

优先级:

高

到期日:

2013-03-20

关闭

创建

图12-25 添加新任务对话框

任务办理-[去XXX客户签订合同]

到期日: 2013-03-20

优先级: 高

创建日期: 2013-03-19 11:38:41

携带资料: 告知书

任务人员

+ 邀请参与人

拥有人: kermit

办理人: kermit

图12-26 创建的新任务

12.4 附件

在流程流转过程中经常会附带一些文件，这些文件由不同的任务办理人上传，一般还会经过多次审批。本节将介绍如何在任务办理过程中创建、上传、读取、删除附件。

一般的企业应用或网站的附件类型为文件，或者一个外部的链接。Activiti对这两种方式都给予了良好的支持。图12-27展示了两种不同类型的附件，第一种为PDF格式的文件，第二种是一个链接类型的附件。



图12-27 任务相关附件

单击图12-27中的“添加”按钮后弹出如图12-28所示的添加附件对话框，针对不同的附件类型用不同的标签页区分，在选择“文件”类型后把选择的文件以二进制流的方式保存到数据库，若选择“Web资源”，则把“URL”的值保存到附件对象（Attachment）的“url”属性中。

引擎针对这两种附件类型分别提供了一个创建附件对象的API（见图12-29），调用了TaskService#createAttachment(attachment)之后可以得到一个附件对象。特别注意的是在该方法被调用之后附件已被持久化到数据库，如果需要更改附件的属性，可以先通过TaskService#getAttachment(attachmentId)查询附件对象，然后设置附件属性（只支持名称和描述属性的更改），之后调用TaskService#saveAttachment(attachment)更新附件到数据库。



图12-28 添加附件

createAttachment
<pre>Attachment createAttachment(String attachmentType, String taskId, String processInstanceId, String attachmentName, String attachmentDescription, InputStream content)</pre>
Add a new attachment to a task and/or a process instance and use an input stream to provide the content
createAttachment
<pre>Attachment createAttachment(String attachmentType, String taskId, String processInstanceId, String attachmentName, String attachmentDescription, String url)</pre>
Add a new attachment to a task and/or a process instance and use an url as the content

图12-29 创建附件的两个API

在图12-29中，两个方法前5个参数的含义都是相同的，可以设置附件对象的类型、任务ID、流程实例ID、附件名称、附件描述信息；唯一不同的是最后一个参数，第一个方法的最后一个参数可以接收一个输入流（InputStream）对象，第二个方法的最后一个参数则接收一个字符串型的URL。

代码清单12-11列出了创建文件类型附件的代码，要创建“Web资源”类型的附件，只要把文件对象换成字符串型URL参数，把附件类型设置为“url”（不再列出具体的代码）。本例的代码见me/kafeitu/activiti/web/chapter12/AttachmentController.java。

代码清单12-11 创建文件类型的附件

```
@RequestMapping(value = "new/file")
public String newFile(@RequestParam("file") MultipartFile file, HttpSession session) {
    String attachmentType = file.getContentType() + ";" + #2
        FilenameUtils.getExtension(file.getOriginalFilename()); #3
    identityService.setAuthenticatedUserId(UserUtil.getUserFromSession(session).getId());
    Attachment attachment = taskService
        .createAttachment(attachmentType, taskId, processInstanceId,
            attachmentName, attachmentDescription, file.getInputStream());
    return "redirect:/chapter6/task/getform/" + taskId;
}
```

代码清单12-11的#1处接收一个“MultipartFile”对象（由Spring MVC提供），考虑到文件类型的附件上传以后要能够下载，并且上传的类型保持一致，所以在保存附件时以Web资源的实际类型作为附件的类型（#2处），例如PDF类型的文件对应“application/pdf”，这样在下载附件时通过设置HttpResponse中头信息的“Content-Type”属性的值为“application/pdf”，浏览器就可以正确解析。

#3处使用了一个工具类来获取文件的扩展名，考虑到从文件对象获取的“ContentType”属性不能直接识别具体类型，例如Excel文件的“ContentType”为“application/octet-stream”，所以在附件类型后面再追加文件的扩展名，在上传一个Excel类型的文件后附件对象的类型为“application/octet-stream; xls”，这样当下载附件时就可以根据这个规则解析，从而把一个二进制流转换成一个文件流输出到浏览器。如何下载文件类型的附件可以参见代码清单12-12。

代码清单12-12 下载附件

```
@RequestMapping(value = "download/{attachmentId}")
public void downloadFile(@PathVariable("attachmentId") String attachmentId, HttpServletResponse response) throws IOException {
    // 查询附件对象
    Attachment attachment = taskService.getAttachment(attachmentId);
    // 读取附件的二进制流
    InputStream attachmentContent = taskService.getAttachmentContent(attachmentId);
    // 根据规则获取文件的类型
    String contentType = StringUtils.substringBefore(attachment.getType(), ";");
    response.setHeader("Content-Type", contentType + ";charset=UTF-8");
    // 根据规则获取文件的扩展名
    String extensionFileName = StringUtils.substringAfter(attachment.getType(), ";");
    // 以附件名字和扩展名作为下载的文件名
    String fileName = attachment.getName() + "." + extensionFileName;
    response.setHeader("Content-Disposition", "attachment; filename=" + fileName);
    IOUtils.copy(new BufferedInputStream(attachmentContent), response.getOutputStream());
}
```

在了解了如何创建、下载附件之后，要下载附件，前提是要把已经上传的附件展示给用户，并且根据附件类型使用不同的处理方式；在上传附件的同时传递了任务ID与流程实例ID，根据不同可以使用不同的方式查询附件。

下面的代码分别根据任务ID和流程实例ID查询附件：

```
List<Attachment> attachments =
    taskService.getTaskAttachments(taskId)
List<Attachment> attachments = taskService
    .getProcessInstanceAttachments(processInstanceId);
```

删除附件与删除任务类似，具体代码如下：

```
taskService.deleteAttachment (attachmentId);
```

12.5 改进意见列表

有些读者可能在上传、删除附件时发现“意见列表”显示不正确，比如图12-30中的“null”。回想一下第9章，意见列表是为了在任务办理时允许办理人、候选人、参与人发表意见。除了用户自行添加的意见之外，引擎还会自动把一些事件记录在意见列表，所以以前的意见列表展示代码就不能胜任了，最终导致意见列表显示异常。本小节将详细介绍有关意见的其他用途。



图12-30 意见列表显示异常

图12-31展示了意见（Comment）对象的数据，字段“TYPE_”值为“comment”表示手动添加的任务办理意见，值为“event”表示由其他操作自动插入作为操作记录存在。字段“ACTION_”记录了每一条记录的动作类型，“AddComment”表示手动添加的意见，“AddAttachment”表示该记录是在创建附件时由引擎自动创建的。此外，当“ACTION_”字段的值为“AddUserLink”或“DeleteUserLink”时，表示添加参与人或者删除参与人由引擎自动插入。“MESSAGE_”字段的值表示了操作动作，例如，当动作为“AddUserLink”且消息为“jenny_candidate”时，表示用户jenny被添加到任务的候选人列表（也可以是自定义的参与人类型，参考12.1.3节）。

SELECT * FROM ACT_HI_COMMENT order by TIME_ desc;								
ID_	TYPE_	TIME_	USER_ID_	TASK_ID_	PROC_INST_ID_	ACTION_	MESSAGE_	FULL_MSG_
1203	event	2013-03-21 23:07:56.317	kermit	22	null	AddUserLink	tom_ _candidate	null
1110	event	2013-03-21 22:02:08.265	kermit	22	null	AddUserLink	jenny_ _candidate	null
1104	event	2013-03-21 19:30:28.568	kermit	22	11	AddAttachment	合同复印件.pdf	null
1027	event	2013-03-21 00:02:08.035	kermit	22	11	AddAttachment	咖啡兔的博客	null
702	comment	2013-03-19 20:18:55.913	kermit	701		AddComment	请补充文件	e694b6e68t

图12-31 ACT_HI_COMMENT表数据

了解了这些数据的来源就可以根据这些类型解决意见列表的“null”问题，可以根据不同的类型使用文字加以说明，下面给出了一些示例。

- **AddComment**：发表了意见。
- **DeleteComment**：删除了意见。
- **AddUserLink**：添加了候选人或者参与人。
- **DeleteUserLink**：删除了候选人或者参与人。
- **AddGroupLink**：添加了候选组。
- **DeleteGroupLink**：删除了候选组。
- **AddAttachment**：添加了附件。
- **DeleteAttachment**：删除了附件。

根据上面的规则，在显示意见时加以处理即可显示如图12-32所示的结果。有一点需要注意，当涉及添加或删除候选人、候选组时流程ID字段（PROC_INST_ID）为空，这样仅根据流程实例ID查询事件列表得不到完整的结果列表，所以本例在读取事件列表时传递了taskId与processInstancelId两个参数，后台会分别查询并把结果合并到一个集合中。

 **说明** 由于现在的“意见列表”显示的数据已经不单单是手动添加的意见了，图12-32中混合了意见、附件、候选人、候选组以及自定义参与人类型的操作事件，所以把“意见”更名为“事件”更为合理。

事件列表			
1.	kermit	添加了附件：咖啡兔的博客	(领导审批) 3/21/2013 12:02:08 AM
2.	kermit	邀请了 jenny 作为[技术顾问]	(领导审批) 3/16/2013 7:46:29 PM
3.	kermit	邀请了 amy 作为[贡献人]	(领导审批) 3/16/2013 7:11:25 PM
4.	kermit	邀请了 amy 作为[候选人]	(领导审批) 3/17/2013 1:28:37 PM
5.	kermit	邀请了 amy 作为[拥有人]	(领导审批) 3/17/2013 1:20:23 AM
6.	kermit	添加了附件：咖啡兔的博客	(领导审批) 3/21/2013 12:02:08 AM
7.	kermit	添加了[候选组] deptLeader	(领导审批) 3/17/2013 1:28:08 PM
8.	kermit	发表了意见：因为王总出差此任务延迟2天	(领导审批) 3/22/2013 12:20:24

图12-32 改进后的意见列表

为了能够兼容不同类型的事件并给出正确合理的中文说明，为此专门写了一个事件转换器，读者可以查看本书配套资源中本章源码的src/main/webapp/js/modules/chapter12/events.js文件了解处理过程，由于篇幅有限，这里不再详细介绍。

12.6 任务委派

任务委派是任务管理范畴内一个比较重要的功能，简答来说就是一个任务本来分配给用户A（委派人），但是A由于某些原因不能处理该任务，可以把这个任务委派给用户B（被委派人）代理，在用户B处理完成后将任务再次回归给用户A。在整个过程中用户A是该任务的拥有人（Owner），用户B是该任务的办理人（Assignee）。图12-33用流程图的方式展示了任务委派的原理。

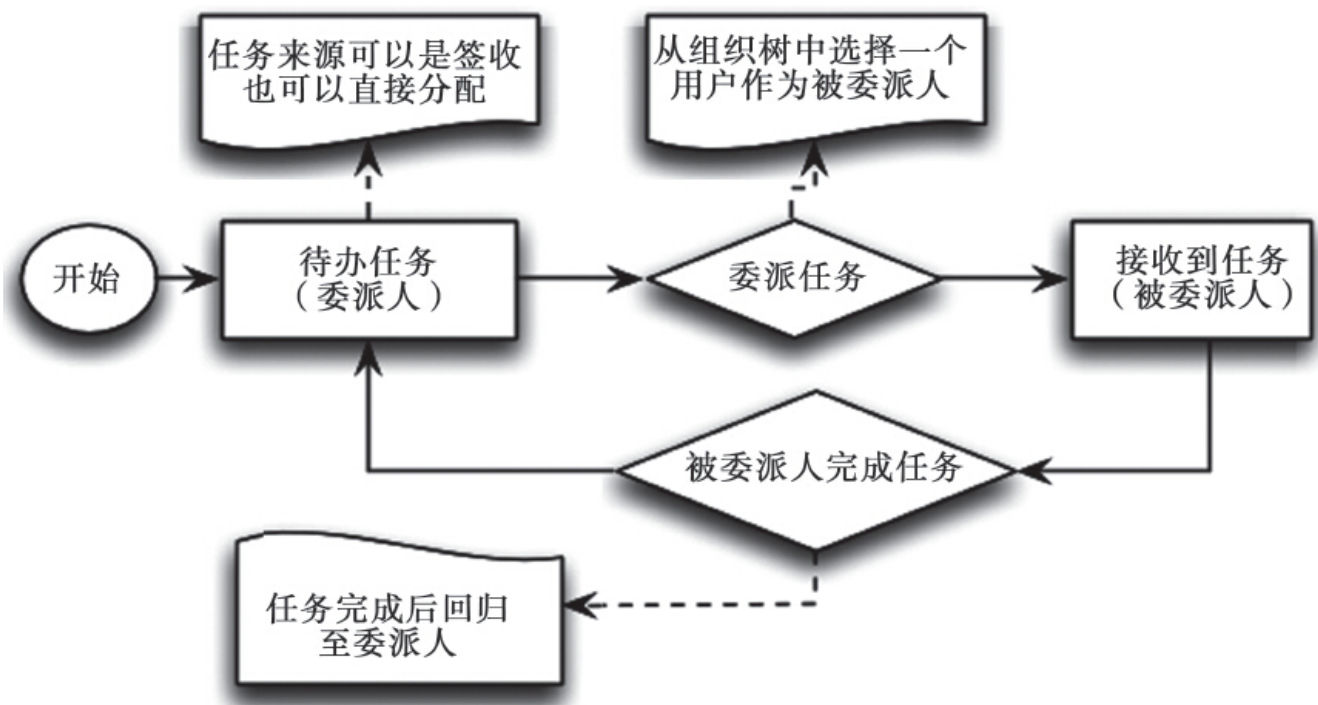


图12-33 任务委派原理

12.6.1 单元测试

引擎提供了专门的API处理任务委派，而且处理被委派的任务时必须使用特定的API才可以。图12-34是一个很简单的流程图，配合代码清单12-13可以展示如何在Activiti中实现任务委派。



图12-34 简单的任务委派流程

图12-34的主要XML代码如下：

```
<process id="taskDelegateProcess" name="任务委派">
  <userTask id="delegatedTask" name="用户任务"
    activiti:assignee="{userId}"></userTask>
</process>
```

代码清单12-13 任务委派单元测试

```
public class TaskDelegateTest extends PluggableActivitiTestCase {
    @Deployment(resources = {"diagrams/chapter12/taskDelegate.bpmn"})
    public void testTaskDelegate() throws Exception {
        Map<String, Object> variables = new HashMap<String, Object>();
        String userId = "bill"; // 委派入
        variables.put("userId", userId);
        runtimeService.startProcessInstanceByKey("taskDelegateProcess", variables);
        Task task = taskService.createTaskQuery().taskAssignee(userId).singleResult();
        assertNull(task.getOwner()); // 第一次被分配任务后拥有人为空
        String delegatedUserId = "henryyan"; // 被委派入
        taskService.delegateTask(task.getId(), delegatedUserId); // 执行委派任务操作 #1
        // 查看数据状态
        task = taskService.createTaskQuery() // 查询被委派且未处理的任务
            .taskDelegationState(DelegationState.PENDING) #2
            .taskAssignee(delegatedUserId).singleResult();
        assertEquals(task.getOwner(), userId); // 委派任务的拥有人为bill
        assertEquals(task.getAssignee(), delegatedUserId); // 委派任务的办理人为henryyan
        // 被委派入处理完成任务
        taskService.resolveTask(task.getId()); // 被委派入完成任务 #3
        // 任务回归到委派入
        task = taskService.createTaskQuery() // 查询被委派且已经处理完成的任务
            .taskDelegationState(DelegationState.RESOLVED) #4
            .taskAssignee(userId).singleResult();
        assertEquals(task.getOwner(), userId); // 被委派的任务完成后拥有人和办理人均为bill
        assertEquals(task.getAssignee(), userId);
        // 委派入完成任务
        taskService.complete(task.getId());
        long count = historyService.createHistoricProcessInstanceQuery().finished().count();
        assertEquals(1, count);
    }
}
```

代码清单12-13用单元测试的方式阐述了任务委派的过程，在#1处调用委派API把bill的待办任务委派给用户henryyan，此时任务的所有人是bill即原任务的办理人；接着在#2处创建任务查询对象查询所有委派给用户henryyan的任务，即委派状态为“pending”的任务，此时任务的拥有人是bill，而办理人是henryyan。

#3处与#1处相对应，被委派入完成被委派的任务，此时拥有人、办理人均为bill，并且该任务的委派状态为“resolved”。

12.6.2 任务表单中的委派

在了解了任务委派的原理及执行过程后，下面把委派功能添加到任务表单中。图12-35中加边框部分是针对任务委派添加的，单击任务委派后的文字可以选择一个委派入，选择委派入之后任务的委派状态将会改变，如图12-36所示，同样在任务列表中也可以看到任务的委派状态，如图12-37所示。

任务办理—[领导审批]， 流程定义ID: [purchase-subprocess:1:10]

到期日： 2013-03-29

优先级： 中

创建日期： 2013-03-14 07:52:11

该任务无描述。

任务人员

+ 邀请参与人

拥有人: bill

办理人: bill

参与人

候选[人|组]

• amy -- 贡献人 (参与人)

✕ 删除

• henry -- 总经理 (参与人)

✕ 删除

• jenny -- 技术顾问 (参与人)

✕ 删除

• deptLeader -- 候选组

✕ 删除

• amy -- 候选人

✕ 删除

• jenny -- 候选人

✕ 删除

• tom -- 候选人

✕ 删除

任务委派:

✓ Amy Zhang

Andy Zhao

Bill Zheng

Eric Li

Henry Yan

Jenny Luo

Kermit Miao

Thomas Wang

Tom Wang

Tony Zhang

图12-35 在任务表单中添加了任务委派功能

任务人员

+ 邀请参与人

拥有人: bill

办理人: bill

任务委派:

✓ 被委派

图12-36 任务委派状态: 被委派

Activiti Explorer

首页 请假 (普通表单) 任务列表 管理

Henry Yan/henry

任务ID	任务名称	流程实例ID	流程定义ID	父任务ID	任务委派状态	任务创建时间	办理人	操作
22	领导审批	11	purchase-subprocess:1:10	无	✓被委派	2013-03-14 07:52:11	henry	办理 反签收

断任务

图12-37 任务列表显示委派状态

任务表单和任务列表中处理任务委派的状态可以参考下面的代码，根据状态显示不同的说明文字。

```
<c:if test="${task.delegationState == 'PENDING' }">
    被委派
</c:if>
<c:if test="${task.delegationState == 'RESOLVED' }">
    任务已处理完成
</c:if>
```

从图12-36中可以看出，当前任务的拥有人是bill，选择的被委派人是“Henry Yan”，从图12-37中也看出用户henry有一条待办任务。

单击任务表单中的“完成任务”按钮，实际调用的是TaskService#complete()方法，但是在被委派的任务完成时必须调用代码清单12-13中#3处的TaskService#resolveTask()方法，否则被委派的任务不能回归给委派人，所以针对这个问题需要在Task-Cotroller#complete()控制器中对任务委派进行特殊处理。

TaskCotroller#complete()首先会查询任务是否存在，解决委派问题可以在其后加入下面的代码，根据任务的委派状态判断，如果处于委派状态，就调用特定的委派回归处理方法，否则按照普通任务处理。

```
if (task.getDelegationState() == DelegationState.PENDING) {
    taskService.resolveTask(taskId);
    return TASK_LIST;
}
```

12.7 本章小结

本章内容是本书的重点，因此花费了大量的篇幅详细介绍和用户任务有关的概念、操作，在一般的流程中大部分活动都是用户任务，是以流程驱动的系统与用户交互的接口。

以Activiti官方Activiti Explorer的任务办理界面为基准，通过6节内容对图12-1的内容——做了详细讲解并实现了全部功能。

通过对任务表单的改造来查看、修改任务的一些属性，例如拥有人、办理人、优先级等；介绍了新增的添加任务相关人员功能，可以动态地添加候选人、候选组、参与人，并对每一种类型的人员做了比较说明；还针对任务查询做了改进，进而可以查询和三种类型人员有关的任务。

子任务是在现有任务的基础上添加的，如果在任务办理过程中需要对任务拆分，可以利用引擎的子任务功能实现。本章带领读者学习了如何创建、完成子任务以及子任务的办理状态。与此类似的还有手动任务，它和子任务的区别在于不依赖现有任务直接创建一个独立的任务。

附件也是本章内容的重点，虽然介绍的篇幅不多但是其在日常使用的重要性不容忽视。本章介绍了可以为办理的任务创建两种不同类型的附件（文件、Web资源），并且演示了如何删除、读取附件内容。

在第9章中简单介绍了利用引擎的Comment实现审批意见功能，不过由于对用户任务的一系列操作“导致”引擎自动产生了一些“意见”，包括对任务相关人员和附件的添加、删除操作记录，所以最终把“意见”更名为“事件”更为恰当。为了能够把计算机能识别的记录转换为文字，还专门写了一个Javascript版本的转换器以供读者参考。

最后介绍了任务委派功能，首先用流程图的形式描述了委派的原理，然后利用一个简单的例子结合单元测试从引擎API的角度去学习，并把这一功能加入到了Activiti Explorer中。

第13章 流程数据查询与跟踪

实战篇是本书比较核心的部分，占据了本书主要的篇幅，在本章之前学习了如何启动流程、完成任务等操作，几乎每个操作都会对数据库产生影响，当然大多数操作都是向数据库中插入数据。一般由流程驱动的系统会提供各种数据的查询功能，例如某个用户参与过的流程、跟踪流程办理状态、已经办理完成（归档）的流程。本章将针对这一系列查询需求进行一一讲解，介绍运行时与历史数据不同的查询方法，以及标准查询API不能满足复杂查询时如何使用Native Query（使用基于MyBatis的SQL语句方式查询）方式自定义查询。

13.1 Query API简介

Activiti的查询API分类两类：标准查询与Native查询。所谓标准查询是在以Java对象的方式通过创建一个指定类型的Query对象（实现Query接口，图13-1的左半部分）后用链式编程的方式设置查询参数。但是标准查询有一个弊端，不支持复杂的查询，比如多张表联合查询或者使用或（SQL的OR关键字）关系查询。因此引擎从5.11版本开始提供了Native查询，允许采用标准SQL的方式查询流程对象，不过Native查询仅支持部分流程对象，并且查询对象需要实现图13-1右半部分的NativeQuery接口。

图13-1中的两个基础查询接口均有一系列特定类型的子接口，图13-2的类图展示了标准查询的子接口，图13-3展示了Native查询的子接口。

Query <...>		NativeQuery <...>	
asc()	T	sql(String)	T
desc()	T	parameter(String, Object)	T
count()	long	count()	long
singleResult()	U	singleResult()	U
list()	List<U>	list()	List<U>
listPage(int, int)	List<U>	listPage(int, int)	List<U>

图13-1 标准查询与Native查询类图



图13-2 标准查询对象继承关系

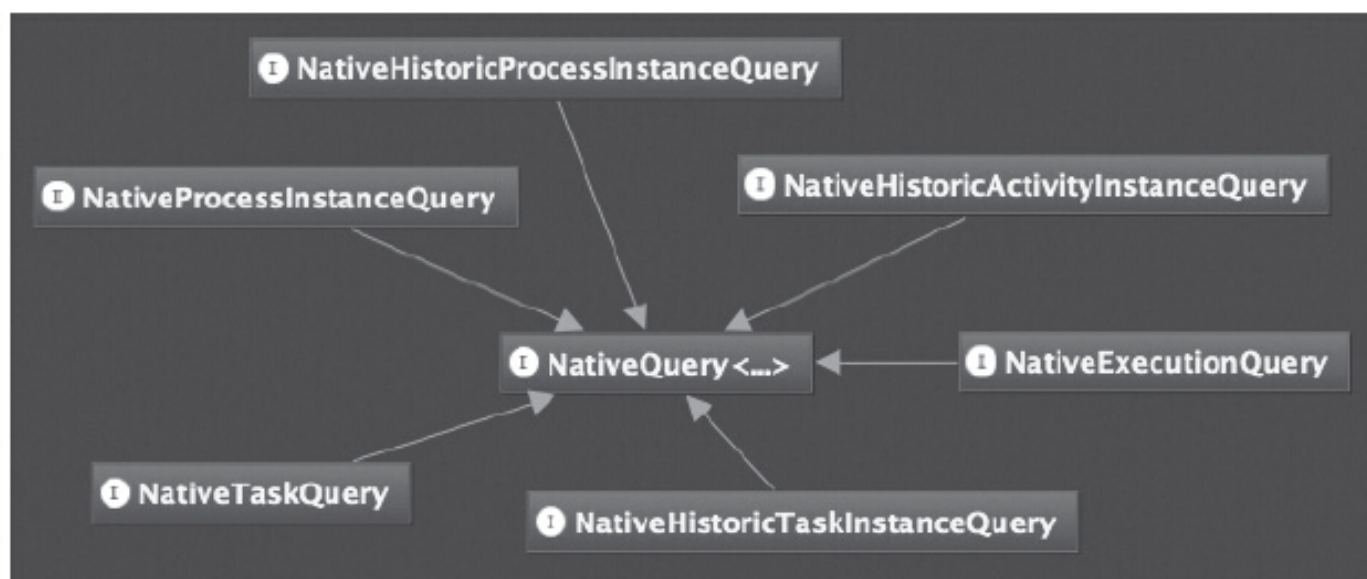


图13-3 Native查询对象继承关系

本节的主要内容是如何查询运行时与历史数据，图13-4（HistoryService）、图13-5（RuntimeService）、图13-6（TaskService）分别列出了创建标准查询对象的API，结合图13-2和图13-3的两个查询对象类图很容易理解Activiti中关于查询的API是如何设计的。

 HistoryService	
 createHistoricProcessInstanceQuery()	HistoricProcessInstanceQuery
 createHistoricActivityInstanceQuery()	HistoricActivityInstanceQuery
 createHistoricTaskInstanceQuery()	HistoricTaskInstanceQuery
 createHistoricDetailQuery()	HistoricDetailQuery
 createHistoricVariableInstanceQuery()	HistoricVariableInstanceQuery
 deleteHistoricTaskInstance(String)	void
 deleteHistoricProcessInstance(String)	void
 createNativeHistoricProcessInstanceQuery()	NativeHistoricProcessInstanceQuery
 createNativeHistoricTaskInstanceQuery()	NativeHistoricTaskInstanceQuery
 createNativeHistoricActivityInstanceQuery()	NativeHistoricActivityInstanceQuery

图13-4 HistoryService中创建查询对象的方法

 RuntimeService	
 createExecutionQuery()	ExecutionQuery
 createNativeExecutionQuery()	NativeExecutionQuery
 createProcessInstanceQuery()	ProcessInstanceQuery
 createNativeProcessInstanceQuery()	NativeProcessInstanceQuery

图13-5 RuntimeService中创建查询对象的方法

 TaskService	
 createTaskQuery()	TaskQuery
 createNativeTaskQuery()	NativeTaskQuery

图13-6 TaskService中创建查询对象的方法

13.2 运行时数据查询

运行时数据查询主要依赖TaskService与RuntimeService这两个接口，在第6章已经介绍了如何查询任务数据，第12章又针对自定义的参与者更改了任务查询方法。本节将再次改造任务查询以支持分页查询，并添加运行中流程查询以使用户跟踪流程办理的进度。

通过前面章节的学习我们知道：一个任务一旦处理完成将流转至下一个节点，而且通常下一个节点的处理人不是当前人，所有参与过流程办理的人可能会再次查看或跟踪流程办理的进度，例如当前任务正在哪个节点由谁办理，甚至可以看到该任务是否被签收（如果有候选人、候选组），还可以查询已经处理的任务的办理情况等再详细一些的信息，例如何时创建任务、何时完成任务等信息。

13.2.1 任务查询

目前的任务列表是把已经签收、未签收、受邀的任务分开查询，然后汇总到一个集合对象中，在实际应用中一般会对任务列表分页查询，很明显目前的查询方式很难做到这一点。

利用Native查询很容易满足分页查询的需求，图13-1中列出了Native查询的几个方法，其中“sql(String sql)”可以接受一个标准的SQL语句，如果需要设置参数，可以用Mybatis的语法设置。下面列出了通过Native查询来查询任务列表的代码。

```
List<Task> tasks = taskService.createNativeTaskQuery()
    .sql("SELECT * FROM " +
        managementService.getTableName(Task.class)
        + " T WHERE T.NAME = #{taskName}")
    .parameter("taskName", "人事审批").list();
```

首先通过TaskService接口创建一个Native方式的查询对象，然后设置SQL语句，其中表名可以通过ManagementService#getTableName()方法获取，条件以Mybatis的格式设置（Native查询方式最终会使用Mybatis的API执行查询），以“#”标注一个参数的开始，然后用“{...}”设置参数名称，最后用链式编程方式调用“parameter()”方法设置每一个参数的值。

获取结果的方式与标准查询相同，方法“list()”可以查询所有结果，方法“list-Page(firstResult, maxResult)”可以分页查询（Activiti 5.13以上的版本支持）。

代码managementService.getTableNames(Task.class)完全可以使用“ACT_RU_TASK”代替，只不过引擎提供了接口可以根据流程对象获取对应的表名。表13-1列出了流程对象与表名的对照关系。

表13-1 流程对象对应的数据库表名

类 名	表 名
org.activiti.engine.task.Task	ACT_RU_TASK
org.activiti.engine.runtime.Execution	ACT_RU_EXECUTION
org.activiti.engine.runtime.ProcessInstance	ACT_RU_EXECUTION
org.activiti.engine.repository.ProcessDefinition	ACT_RE_PROCDEF
org.activiti.engine.repository.Deployment	ACT_RE_DEPLOYMENT
org.activiti.engine.runtime.Job	ACT_RU_JOB
org.activiti.engine.history.HistoricProcessInstance	ACT_HI_PROCIINST
org.activiti.engine.history.HistoricActivityInstance	ACT_HI_ACTINST
org.activiti.engine.history.HistoricDetail	ACT_HI_DETAIL
org.activiti.engine.history.HistoricVariableUpdate	ACT_HI_DETAIL
org.activiti.engine.history.HistoricFormProperty	ACT_HI_DETAIL
org.activiti.engine.history.HistoricTaskInstance	ACT_HI_TASKINST
org.activiti.engine.history.HistoricVariableInstance	ACT_HI_VARINST

代码清单13-1 使用Native Query分页查询

```
Page<Task> page = new Page<Task>(PageUtil.PAGE_SIZE);
int[] pageParams = PageUtil.init(page, request);
String sql = "select distinct RES.* from ACT_RU_TASK RES left join
    \"ACT_RU_IDENTITYLINK I on I.TASK_ID = RES.ID WHERE \" +
    \" ( RES.ASSIGNEE = #{userId}\" + // 已经签收或者直接分配的 #1
    \" or (RES.ASSIGNEE is null and ( I.USER_ID = #{userId} or I.GROUP_ID in (select
    G.GROUP_ID from ACT_ID_MEMBERSHIP G where G.USER_ID_ = #{userId} ) ) )\" +
    \" )\" + filters + \" order by RES.CREATE TIME desc\";
NativeTaskQuery query = taskService.createNativeTaskQuery().sql(sql)
    .parameter(\"userId\", user.getId());
List<Task> tasks = query.listPage(pageParams[0], pageParams[1]);
page.setResult(tasks);
page.setTotalCount(query.sql(\"select count(*) from (\" + sql + \")\").count());
// 统计总数
mav.addObject(\"page\", page);
```

代码清单13-1对两种类型的任务进行了合并查询，其结果和之前的标准查询一致，查询后的结果如图13-7所示。#1处语句对应如下的标准查询：

```
taskService.createTaskQuery().taskAssignee(userId);
```

#1处后面的语句对应如下的标准查询，结果包含了候选人、候选组、其他类型的参与人范围内的任务。

```
taskService.createTaskQuery().taskInvolvedUser(userId)
```



图13-7 通过NativeQuery分页查询任务

当然在图13-7中的列表还可以添加根据任务名称或者其他属性的过滤条件，如图13-8所示，在“任务名称”中添加了查询条件，过滤出任务名称中包含“合同”的待办任务。

任务名称：

合同

查询

任务ID	任务名称	流程实例ID	流程定义ID	父任务ID	任务委派状态	任务创建时间	办理人	操作	新任务
801	去XXX客户签订合同			无		2013-03-19 11:38:41	kermit	办理 反签收	

共1条数据<<<1>>>

图13-8 在任务列表中添加过滤条件

在代码清单13-1的基础上加以改进即可实现条件过滤，修改后的代码如下：

```
NativeTaskQuery query = taskService.createNativeTaskQuery();
String filters = "";
if (StringUtils.isNotBlank(taskName)) {
    filters += " and RES.NAME like #{taskName}";
    query.parameter("taskName", "%" + taskName + "%");
    mav.addObject("taskName", taskName);
}
String assignedSql = "select http://www.hzcourse.com/resource/readBook?path=/openresources/teach_ebook/uncompressed/15030/OEBPS/Text/... " + filters;
String claimOrLinkedSql = "select http://www.hzcourse.com/resource/readBook?path=/openresources/teach_ebook/uncompressed/15030/OEBPS/Text/... " + filters;
```

这样，在图13-8中的“任务名称”输入框中输入“合同”文字后再查询就可以过滤出任务名称包含“合同”的任务。

13.2.2 查询参与的流程

在本节开始的时候提出了一个需求，查询已经办理过的流程以及跟踪流程办理情况。在前面的章节中已经多次使用标准查询来查询流程运行中的流程实例，代码如下：

```
List<ProcessInstance> processInstanceList =
    runtimeService.createProcessInstanceQuery().list();
List<Execution> executionList =
    runtimeService.createExecutionQuery().list();
```

在第9章中已经介绍过ProcessInstance与Execution的不同了，两者是一对多的关系，所以processInstanceList的结果数量会“大于等于” executionList的结果数量。

查看ProcessInstanceQuery与ExecutionQuery对象的API没有直接查询某个用户参与过的流程，所以需要通过Native查询实现。图13-9展示了新添加的“参与的流程”列表，自定义的SQL如下：

```
NativeExecutionQuery nativeExecutionQuery =
    runtimeService.createNativeExecutionQuery();
String sql = "select RES.* from ACT_RU_EXECUTION RES left join
    ACT_HI_TASKINST ART on ART.PROC_INST_ID =
    RES.PROC_INST_ID
    where ART.ASSIGNEE = #{userId} and ACT_ID is not null
    and IS_ACTIVE_ = 'TRUE' order by START_TIME_desc";
```

以上代码位于me.kafeitu.activiti.web.chapter13.ExecutionController.java文件中(见代码清单13-2)，JSP文件位于chapter13/src/main/webapp/WEB-INF/views/chapter13/execution-list.jsp。

Activiti Explorer

首页 请假（普通表单） 运行中 管理

Kermit Miao/kermit

任务列表 参与的流程

执行ID	流程实例ID	流程定义ID	当前节点
11	11	purchase-subprocess:1:10	deptLeaderAudit

共1条数据<<<1>>>

图13-9 参与的流程列表

注意，图13-9中“当前节点”列显示的数据与设计流程时的任务ID相同，且仅通过流程定义ID用户不能区分出是哪一个类型的流程（用户体验非常差），针对这两点需要做一些“翻译”的工作，把计算机所表达的数据翻译成用户能看懂的文字，效果如图13-10所示。

plorer

首页 请假（普通表单） 运行中 管理

Kermit Miao/kermit

执行ID	流程实例ID	所属流程	流程定义ID	当前节点
1760	1760	办公用品采购	purchase-subprocess:2:1316	财务审批（未签收）普通流程
1722	1722	办公用品采购—callactivity方式	purchase-callactivity:2:1314	通用付款流程 财务审批（未签收）调用活动
1626	1601	请假流程-会签	leave-countersign:1:1319	[部门/人事]联合会签（办理中kermit）子流程

共3条数据<<<1>>>

图13-10 在参与的流程列表中翻译了当前节点名称

根据当前处于活动的任务类型划分为普通流程、调用活动、子流程。普通流程类型的Execution对象的“activityId”值为在流程定义中定义的活动ID。如果子流程处于活动状态，则流程实例

的“activityId”值为空且活动状态为“FALSE”（表ACT_RU_EXECUTION中的IS_ACTIVE 字段），由其派生的Execution对象处于活动状态并通过字段“PARENT_ID ”与流程实例建立关系。调用活动与子流程类似但也有不同，如果调用活动处于活动状态，可以根据当前流程的流程ID查询相关子流程。图13-10中第二条记录是一个“办公用品采购-callactivity方式”流程，当前节点为“通用付款流程->财务审批”表示当前活动的任务属于“通用付款流程”。

代码清单13-2中使用内部的非公开类ExecutionEntity，目的是获取当前的活动ID，但是在Activiti 5.13版本之前直接通过Execution接口不能获取，需要强制类型转换为ExecutionEntity后再获取，或者通过反射的方式来获取。

代码清单13-2 查询参与的流程列表
<pre>List<Execution> executionList = nativeExecutionQuery.sql(sql) // 分页查询 .listPage(pageParams[0], pageParams[1]); // 查询流程定义对象 Map<String, ProcessDefinition> definitionMap = new HashMap<String, ProcessDefinition>(); // 任务的英文-中文对照 Map<String, Task> taskMap = new HashMap<String, Task>(); // 每个Execution的当前活动ID, 可能为多个 Map<String, List<String>> currentActivityMap = new HashMap<String, List<String>>(); // 设置每个Execution对象的当前活动节点 for (Execution execution : executionList) { ExecutionEntity executionEntity = (ExecutionEntity) execution; String processInstanceId = executionEntity.getProcessInstanceId(); String processDefinitionId = executionEntity.getProcessDefinitionId(); // 缓存ProcessDefinition对象到Map集合 definitionCache(definitionMap, processDefinitionId); // 查询当前流程的所有处于活动状态的活动ID, 如果流程中存在并行任务且此时处于活动状态的 // Activity有多个, 则查询出来的结果是一个集合 List<String> activeActivityIds = runtimeService.getActiveActivityIds(execution.getId()); currentActivityMap.put(execution.getId(), activeActivityIds); for (String activityId : activeActivityIds) { // 查询处于活动状态的任务 Task task = taskService.createTaskQuery().taskDefinitionKey(activityId) .executionId(execution.getId()).singleResult(); if (task == null) { // 处理调用活动 ProcessInstance processInstance = runtimeService.createProcessInstanceQuery() .superProcessInstanceId(processInstanceId).singleResult(); task = taskService.createTaskQuery() // 查询调用活动的当前节点 .processInstanceId(processInstance.getProcessInstanceId()).singleResult(); definitionCache(definitionMap, processInstance.getProcessDefinitionId()); } taskMap.put(activityId, task); // 用Execution的活动ID作为KEY, 以便获取当前任务 } } mav.addObject("taskMap", taskMap); mav.addObject("definitions", definitionMap); mav.addObject("currentActivityMap", currentActivityMap);</pre>

图13-10中显示的“所属流程”就可以通过下面的代码显示出来，从definitions缓存对象中读取流程定义对象，然后获取流程定义的名称，即流程名称。

<td>\${definitions[e.processDefinitionId].name}</td>
--

图13-10中的“当前节点”列原本显示的是英文名称，用代码清单13-3所示的代码处理节点名称即可显示为中文的任务名称，并且可以显示每个任务的办理人以及状态。

代码清单13-3 处理当前节点显示逻辑
<pre><c:forEach items="\${currentActivityMap[e.id]}" var="acid"> <c:set var="task" value="\${taskMap[acid]}" /> // 根据活动ID获取Task对象 #1 <!-- 处理 [调用活动] --> <c:if test="\${task.processDefinitionId != e.processDefinitionId}"> \${definitions[task.processDefinitionId].name} </c:if> \${task.name} <c:if test="\${empty task.assignee}">(未签收) </c:if> <c:if test="\${not empty task.assignee}"> (办理中\${task.assignee}) </c:if> </c:forEach></pre>

13.3 流程图跟踪

13.2.2节介绍了如何查询和当前用户相关的流程实例，但是仅查看当前活动的节点还不能满足需求，比如用户要跟踪流程办理的每一步的信息，再比如任务何时创建、何时完成、任务的办理人等信息。

图形是最直观的展示方式和记忆方式，除了能跟踪刚刚提到的信息之外，能以图形的方式展示每一个流程中处于活动的任务和已经办理过的任务，以不同的颜色对不同状态的任务加以区分（蓝色表示已经办理过，红色表示当前节点），这样可以大幅度提升用户体验。图形化的流程跟踪如图13-11所示，其中“[部门/人事]联合会签”为当前节点使用红色边框表示。

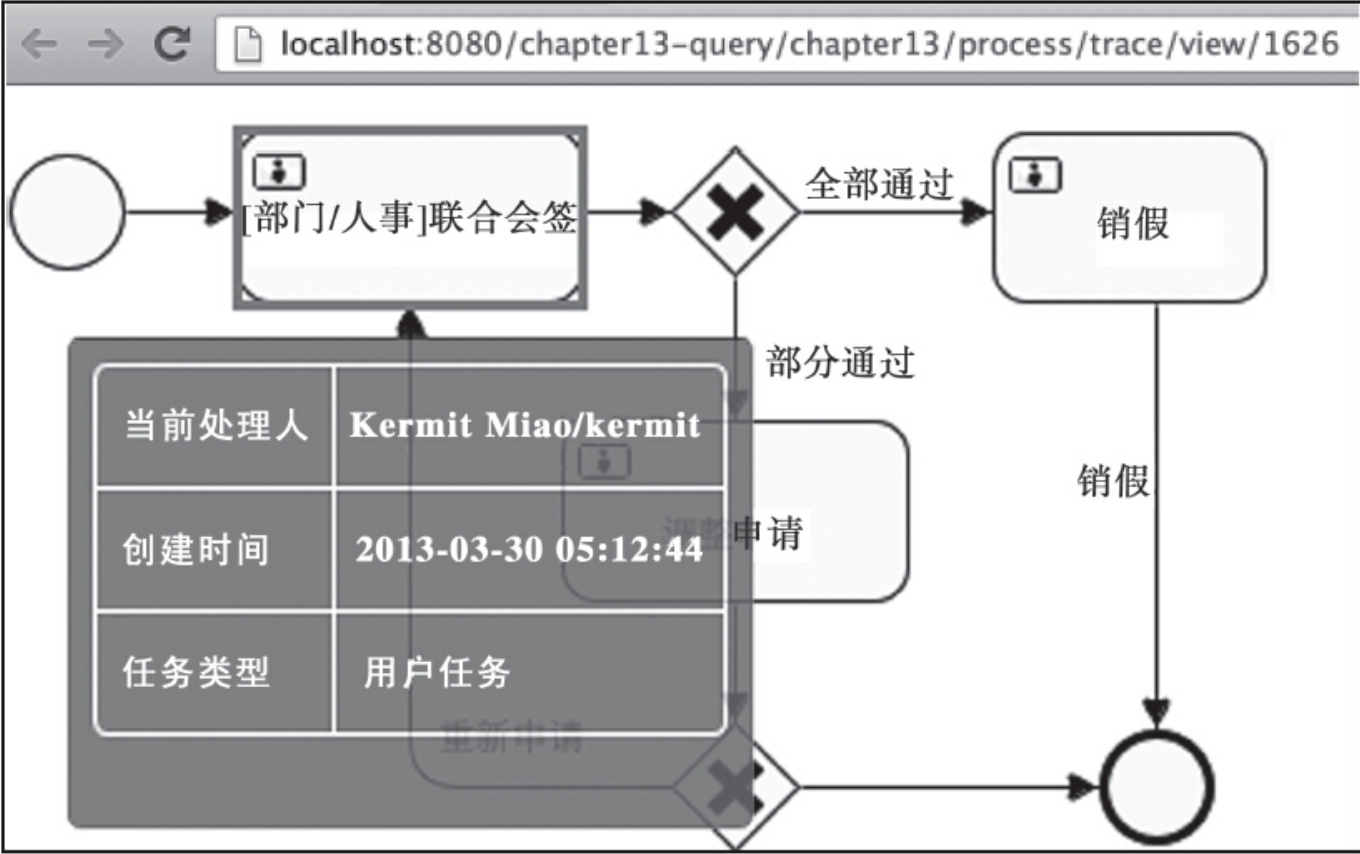


图13-11 图形化流程跟踪

图形化流程跟踪也会遇到和13.2.2节处理当前节点时相同的问题，需要根据三种不同类型的数据状态获取当前活动ID，这样才能在展示层根据状态用红色边框标注当前处于活动的节点。

本节的代码位于webapp/WEB-INF/views/chapter13/trace-process.jsp以及src/main/java/me/kafeitu/web/chapter13/TraceProcessController.java中。

通过代码清单13-3中#1处的链接可以打开流程跟踪的页面，图13-11中的URL表示跟踪ID为1626的Execution对象的办理状态。

要实现图形化流程跟踪首先需要读取流程的图片文件，在第5章中已经介绍了如何根据流程定义ID读取图片资源，所以本节不再重复介绍，读者可以参考相关示例代码。

如何定位当前节点在流程图中的位置是本节的核心内容，获取这些信息需要使用引擎的内部API，这样可以根据流程定义ID获取整个流程定义中所有活动（Activity）集合，然后利用当前流程中处于活动状态的活动ID与流程定义中的活动对象的ID匹配，从而决定红色边框定位在何处。

代码清单13-4列出了利用内部API获取流程定义中所有活动对象的代码。

代码清单13-4 利用内部API获取流程定义中所有活动对象

```
RepositoryServiceImpl repositoryServiceImpl = (RepositoryServiceImpl) repositoryService;
ReadOnlyProcessDefinition deployedProcessDefinition = repositoryServiceImpl
    .getDeployedProcessDefinition(executionEntity.getProcessDefinitionId());
ProcessDefinitionEntity processDefinition =
    (ProcessDefinitionEntity) deployedProcessDefinition;
List<ActivityImpl> activitiList = processDefinition.getActivities();
```

代码清单13-4最终获取到了ActivityImpl集合，该集合中的每一个对象都有一些了getter方法，例如getWidth()、getHeight()、getX()、getY()分别获取了宽度、高度、XY坐标信息。当然也可以通过调用该对象的getProperties().get("type")方法获取每个活动的实际类型，例如UserTask表示用户任务，startEvent表示启动事件等。

了解了ActivityImpl对象的功能之后就可以利用循环的方式处理每个对象，根据活动类型的不同以及是否是当前节点把数据封装到一个Map对象集合中传递给展示层（本例使用JSON方式共展示层通过Ajax方式获取）。循环处理代码见代码清单13-5。

代码清单13-5 循环处理ActivityImpl封装响应数据

```
// 最终返回给展示层的结果变量
List<Map<String, Object>> activityInfos = new ArrayList<Map<String, Object>>();
for (ActivityImpl activity : activitiList) {
    // 不同的活动类型对应不同的Behavior
    ActivityBehavior activityBehavior = activity.getActivityBehavior();
    boolean currentActiviti = false;
    if (activity.getId().equals(activityId)) { // 判断是否为当前节点
        currentActiviti = true;
    }
    // 如果不是当前节点且活动类型属于子流程，或者将调用活动设置为当前节点
    // 这里为了展示不做过多的复杂处理，读者可以在此基础上加以改进
    if (!currentActiviti && (activityBehavior instanceof SubProcessActivityBehavior
        || activityBehavior instanceof CallActivityBehavior)) {
        currentActiviti = true;
    }
    Map<String, Object> activityImageInfo =
        packageSingleActivitiInfo(activity, executionEntity, currentActiviti);
    activityInfos.add(activityImageInfo);
}
```

代码清单13-5最后调用的是packageSingleActivitiInfo()方法，用于根据ActivityImpl对象的属性设置相关信息来为展示层提供数据。packageSingleActivitiInfo()方法见代码清单13-6。

代码清单13-6 packageSingleActivitiInfo()方法

```
private Map<String, Object> packageSingleActivitiInfo(ActivityImpl activity,
    ExecutionEntity execution, boolean currentActiviti) throws Exception {
    Map<String, Object> vars = new HashMap<String, Object>();
    Map<String, Object> activityInfo = new HashMap<String, Object>();
    activityInfo.put("currentActiviti", currentActiviti); // 设置当前节点标志
    activityInfo.put("width", activity.getWidth()); // 设置在图片中的宽度
    activityInfo.put("height", activity.getHeight()); // 设置在图片中的高度
    activityInfo.put("x", activity.getX()); // 设置在图片中的x坐标
    activityInfo.put("y", activity.getY()); // 设置在图片中的y坐标
    Map<String, Object> properties = activity.getProperties();
    // 把活动类型的英文转换为中文
    vars.put("任务类型", ActivityUtil.getZhActivityType(properties.get("type").toString()));
    // 当前节点的task
    if (currentActiviti) {
        Task currentTask = taskService.createTaskQuery().executionId(execution.getId())
            .taskDefinitionKey(execution.getActivityId()).singleResult();
        if (currentTask == null) return;
        String assignee = currentTask.getAssignee();
        if (assignee != null) {
            User assigneeUser = identityService.createUserQuery()
                .userId(assignee).singleResult();
            String userInfo = assigneeUser.getFirstName() + " " + assigneeUser.getLastName()
                + "/" + assigneeUser.getId();
            vars.put("当前处理人", userInfo);
            vars.put("创建时间", currentTask.getCreateTime());
        } else { vars.put("任务状态", "未签收"); }
    }
    activityInfo.put("vars", vars);
    return activityInfo;
}
```

后台代码准备就绪，在展示层中添加如下的代码来读取流程图。

```

```

然后JavaScript通过Ajax请求获取JSON数据循环处理，即可得到如图13-12所示的效果。对于JavaScript如何实现定义涉及JavaScript的一些用法，这里不做详细解释，读者可以参考本章源码的trace-process.js文件，该文件做了一个简单的示例，读者可以使用自己熟悉的展示工具或插件实现相同功能。

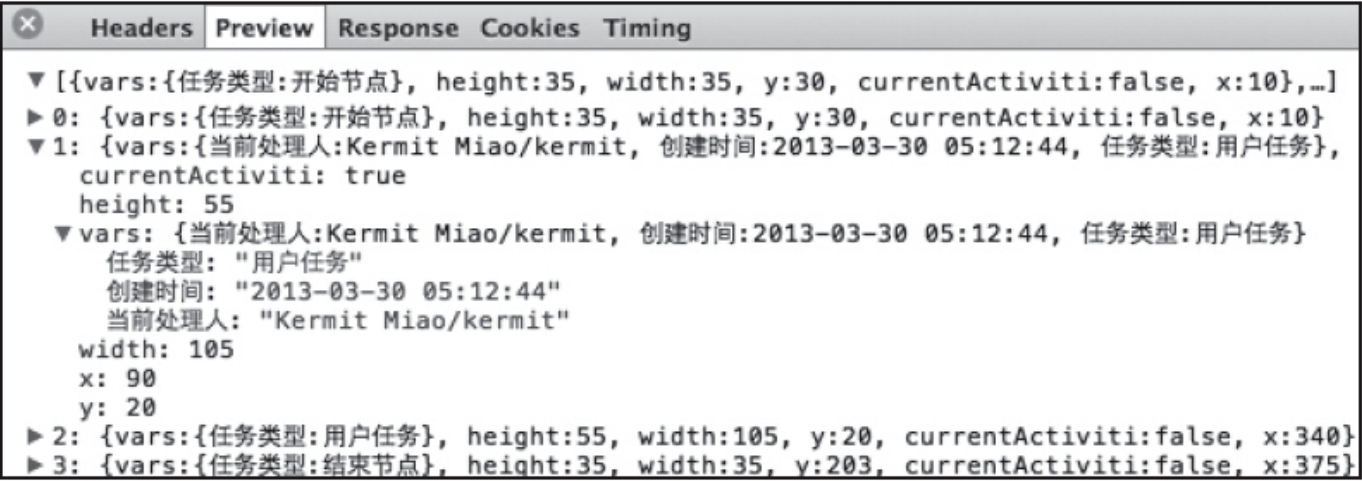


图13-12 读取JSON格式的流程跟踪数据

为了实现图形化流程跟踪，不仅要编写Java代码还需要展示层的协助（JavaScript代码），有没有比较简单的方式呢？我们知道，如果在部署流程时部署了XML（扩展名为bpmn或bpmn20.xml）类型的文件，引擎会使用ProcessDiagramGenerator工具类生成PNG格式的图片作为流程图，这个工具类允许在生成图片的时候用红色边框标记某些节点，最终生成的图片与图13-11中的红色边框一样可以标注当前的活动节点。

代码清单13-7列出了控制层代码。

代码清单13-7 引擎内置的图形化流程跟踪

```
@RequestMapping(value = "trace/data/auto/{executionId}")
public void readResource(@PathVariable("executionId") String executionId,
    HttpServletResponse response) throws Exception {
    ProcessInstance processInstance = runtimeService.createProcessInstanceQuery()
        .processInstanceId(executionId).singleResult();
    BpmnModel bpmnModel = repositoryService // 根据流程定义ID获取BpmnModel
        .getBpmnModel(processInstance.getProcessDefinitionId()); #1
    List<String> activeActivityIds = runtimeService.getActiveActivityIds(executionId); #2
    InputStream imageStream = ProcessDiagramGenerator.generateDiagram(bpmnModel, #3
        "png", activeActivityIds);
    // 输出资源内容到相应对象
    byte[] b = new byte[1024];
    int len;
    while ((len = imageStream.read(b, 0, 1024)) != -1) {
        response.getOutputStream().write(b, 0, len);
    }
}
```

首先查询流程实例对象获取流程定义ID，然后在#1处获取一个BpmnModel对象（封装了流程文件中的所有Activity对象），在#2处查询当前Execution对象的所有处于活动状态的Activity ID集合，把这个集合用#3处的ProcessDiagramGenerator工具类生成图片文件流对象。

在代码清单13-7中，#3处的代码在Activiti 5.16版本不在适用，从该版本开始把图片生成功能提取为一个独立的模块（activiti-image-generator），ProcessDiagramGenerator类也从class转换为interface，原来的ProcessDiagramGenerator类被重新包装命名为DefaultProcessDiagramGenerator作为引擎默认的图片生成器，这样的改变允许我们自定义自己的流程图生成器并注入引擎中。自定义的流程图生成器实现ProcessDiagram-Generator接口并在引擎配置对象中设置“processDiagramGenerator”属性，相关配置如下：

```
<bean id="processEngineConfiguration" class="org.activiti.xxx.XxxProcess-EngineConfiguration">
    http://www.hzcourse.com/resource/readBook?path=/openresources/teach_ebook/uncompressed/15030/OEBPS/Text/...
    <property name="processDiagramGenerator">
        <bean class="me.kafetu.activiti
            chapter13.MyProcessDiagramGenerator" />
    </property>
</bean>
```

最后在展示层读取图片流即可看到如图13-13所示的流程图，代码如下：

```

```

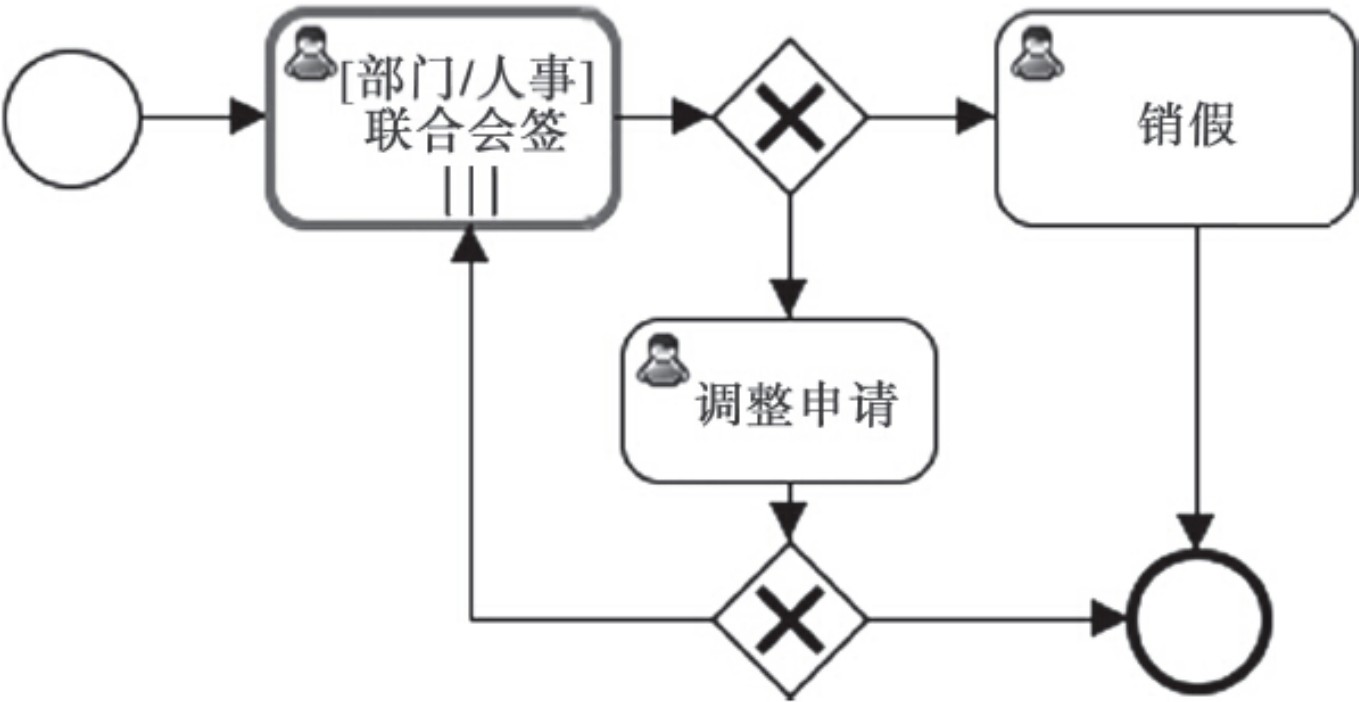


图13-13 使用引擎提供的工具类生成的流程图并标注了当前节点

如果使用引擎默认的配置，在Windows和Linux平台下引擎自动生成的图片会有中文乱码问题，也就是第5章中介绍的那样，只能通过修改字体文件的方式解决这个问题。从Activiti 5.12版本开始彻底解决了这个问题，引擎允许通过配置的方式设置生成流程图时使用的字体，在引擎中添加如下的配置即可解决乱码问题。

```
<property name="activityFontName" value="宋体" />
```

虽然使用引擎提供的方式可以很容易获取到标记了活动节点的流程图，但是这种方式也有不足之处：

- 未使用原流程文件中的坐标信息定位活动。
- Flow上的文字未显示（因为不同设计器的Flow文字的坐标算法不同导致错位或不显示，未来版本会修复）。
- 展示层很难在此基础上扩展功能，因为坐标信息与ActivityImpl对象中的不同。

13.4 历史数据查询

历史数据查询是相对于运行时数据来说的，查询方式与之类似，可以使用标准查询，也可以使用Native查询。本章开头的图13-2、图13-3中以“Historic”开头的查询对象都属于历史数据范畴。

13.4.1 查询历史活动及表单

13.3节用图形方式直观地展示了当前流程的活动节点，如果此时用户需要查询该流程的办理过程，例如哪个节点由谁办理或者相关的变量等信息时就需要通过HistoryService接口查询，在图13-11的页面中再添加图13-14中三个表格的历史数据就可以满足需求了。图13-14中的历史数据包含了流程综合信息（HistoricProcessInstance）、活动记录（HistoricActivityInstance）、相关变量（HistoricVariableInstance），对应的API调用可以参考代码清单13-8。

流程综合信息-【办公用品采购】

流程ID	1760	流程定义ID	purchase-subprocess:2:1316	业务KEY	
流程启动时间	2013-03-30 09:30:08	流程结束时间		流程状态	未结束

活动记录

活动ID	活动名称	活动类型	任务ID	办理人	活动开始时间	活动结束时间	活动耗时（秒）
1763	startevent1	startEvent			2013-03-30 09:30:08	2013-03-30 09:30:08	0.0
1770	领导审批	userTask	1771	kermit	2013-03-30 09:30:08	2013-03-30 09:30:29	20.78
1775	Exclusive Gateway	exclusiveGateway			2013-03-30 09:30:29	2013-03-30 09:30:29	0.0
1776	联系供货方	userTask	1777	thomas	2013-03-30 09:30:29	2013-03-30 09:31:13	43.976
1790	付费子流程	subProcess			2013-03-30 09:31:13		0.0
1791	财务审批	userTask	1792		2013-03-30 09:31:13		0.0

相关变量

变量名称	变量类型	值
applyUserId	string	kermit
dueDate	date	Fri Apr 05 00:00:00 CST 2013
listing	string	sdfrtw
amountMoney	double	32342.0
deptLeaderApproved	string	true
supplier	string	苹果公司
bankName	string	中国工商银行

图13-14 历史相关数据

代码清单13-8 查询历史活动及表单数据

```
// 查询所有的历史活动记录
List<HistoricActivityInstance> activityInstances = historyService
    .createHistoricActivityInstanceQuery().processInstanceId(processInstanceId).list();
// 查询历史流程实例
HistoricProcessInstance historicProcessInstance = historyService
    .createHistoricProcessInstanceQuery().processInstanceId(processInstanceId).singleResult();
// 查询流程有关的变量
List<HistoricVariableInstance> variableInstances = historyService
    .createHistoricVariableInstanceQuery().processInstanceId(processInstanceId).list();
```

需要特别说明的是，如果查询了HistoricActivityInstance类型的数据，就没有必要再查询HistoricTaskInstance，因为前者包含了后者，图13-14中第二个表格的“活动类型”值为“userTask”的数据即表示一个用户任务的办理过程，“任务ID”列也显示了相关任务的ID。

另外，如果要查询任务表单产生的数据需要查询HistoricDetail对象，通过下面的代码可以只查询与流程相关的表单属性，结果如图13-15所示。查询到的结果的实际类型为org.activiti.engine.history.HistoricFormProperty，可以通过getPropertyId()和getPropertyValue()获取表单属性的名称与值。

```
List<HistoricDetail> formProperties = historyService
    .createHistoricDetailQuery().formProperties().list();
```


表单属性

属性名称	属性值	任务ID	设置时间
amountMoney	32342		2013-03-30 09:30:08
listing	sdfwfw		2013-03-30 09:30:08
dueDate	2013-04-05		2013-03-30 09:30:08
deptLeaderApproved	true	1771	2013-03-30 09:30:29
planDate	2013-04-05	1777	2013-03-30 09:31:13
bankName	中国工商银行	1777	2013-03-30 09:31:13
supplier	苹果公司	1777	2013-03-30 09:31:13
bankAccount	622232947927342	1777	2013-03-30 09:31:13

图13-15 流程相关表单属性

13.4.2 查询已归档流程

图13-16展示了“已归档”列表，也就是已经结束的流程实例，可以看到列表中显示了流程的启动、结束时间以及流程启动人等信息，如果该流程是由其他流程派生的，则显示“父流程ID”属性值。

Activiti Explorer

首页

请假（普通表单）

运行中

已归档

管理

Kermit Miao/kermit

流程实例ID	所属流程	流程定义ID	启动时间	流程启动人	结束时间	父流程ID
11	办公用品采购	purchase-subprocess:1:10	2013-03-14 07:52:11	kermit	2013-04-03 10:01:29	
1722	办公用品采购--callactivity方式	purchase-callactivity:2:1314	2013-03-30 08:25:58	thomas	2013-04-03 10:03:32	
1752	通用付款流程	payment:2:1315	2013-03-30 08:40:10	thomas	2013-04-03 10:00:55	1722
1760	办公用品采购	purchase-subprocess:2:1316	2013-03-30 09:30:08	kermit	2013-04-03 10:03:06	

共4条数据<<<1>>>

图13-16 已归档的流程实例

单击流程实例ID可以打开图13-14所示的页面，可以查看该流程办理过程中产生的历史记录。代码清单13-9列出了查询该列表的代码，并启用了分页查询。

代码清单13-9 查询已归档流程实例

```
@RequestMapping(value = "finished/list")
public ModelAndView finishedProcessInstanceList(HttpServletRequest request) {
    ModelAndView mav = new ModelAndView("chapter13/finished-process");
    Page<HistoricProcessInstance> page = new
        Page<HistoricProcessInstance>(PageUtil.PAGE_SIZE);
    int[] pageParams = PageUtil.init(page, request);
    HistoricProcessInstanceQuery historicProcessInstanceQuery = historyService
        .createHistoricProcessInstanceQuery().finished(); // 只查询已经结束的数据
    List<HistoricProcessInstance> historicProcessInstances = historicProcessInstanceQuery
        .listPage(pageParams[0], pageParams[1]);
    // 查询流程定义对象
    Map<String, ProcessDefinition> definitionMap = new HashMap<String, ProcessDefinition>();
    for (HistoricProcessInstance historicProcessInstance : historicProcessInstances) {
        definitionCache(definitionMap, historicProcessInstance.getProcessDefinitionId());
    }
    page.setResult(historicProcessInstances);
    page.setTotalCount(historicProcessInstanceQuery.count()); // 统计总数
    mav.addObject("page", page);
    mav.addObject("definitions", definitionMap);
    return mav;
}
```

13.5 基于MyBatis的CustomSql查询

标准查询与Native查询各有优势，前者允许以链式编程的方式查询数据，而Native查询方式可以自由地拼接SQL语句组合查询。

对于简单查询来说，这两种查询基本能满足需求，但是对于稍微复杂一点的需求就难以胜任了，例如查询当前正在运行的任务并根据流程定义名称进行过滤，显示的字段如图13-17所示。

← → ↻

localhost:8080/chapter13-query/main/index#

☆

Activiti Explorer

🏠 首页

📄 请假（普通表单）

🔄 运行中

📁 已归档

🔧 管理

👤 Kermit Miao/kermit

流程名称：

全部

查询

流程定义

运行中任务

任务ID	任务名称	办理人	流程实例ID	流程名称	流程定义ID	创建时间
1640	[部门/人事]联合会签	andy	1601	请假流程-会签	leave-countersign:1:1319	Sat Mar 30 17:12:44 CST 2013
1649	[部门/人事]联合会签	henry	1601	请假流程-会签	leave-countersign:1:1319	Sat Mar 30 17:12:44 CST 2013

图13-17 运行中任务列表

图13-17除了包含任务的主要属性外还包含“流程名称”字段，如果使用标准查询，需要先查询相关任务再查询与任务相关联的流程定义来获取“流程名称”属性。使用Native查询虽然可以编写SQL语句来关联查询，但是Native查询只能返回引擎的几个固定对象，也就是返回结果不能自定义。

为了解决查询的灵活性问题，从Activiti 5.15版本开始添加了基于MyBatis的查询接口，允许开发人员使用MyBatis的语法查询数据，并且查询结果的类型很灵活，该功能被封装在ManagementService接口的executeCustomSql方法中。

图13-18列出了executeCustomSql的API，从中可以看出该方法接收一个类型为CustomSqlExecution的参数，并且需要泛型<MapperType, ResultType>：MapperType表示定义MyBatis语法查询的接口类型，ResultType表示返回结果的类型。代码清单13-10列出了查询图13-17中列表的相关代码，从代码中可以看出CustomSqlExecution接口泛型为<TaskQueryMapper, List<RunningTask>>。

<MapperType, ResultType>
ResultType

executeCustomSql(CustomSqlExecution<MapperType, ResultType> customSqlExecution)
[EXPERIMENTAL] Executes the sql contained in the CustomSqlExecution parameter.

图13-18 ManagementService的executeCustomSql方法API

代码清单13-10 使用CustomSql查询运行中任务

```
CustomSqlExecution<TaskQueryMapper, List<RunningTask>> customSqlExecution =
    new AbstractCustomSqlExecution<TaskQueryMapper,
        List<RunningTask>>(TaskQueryMapper.class) {
    public List<RunningTask> execute(TaskQueryMapper customMapper) {
        List<Map<String, Object>> maps = null;
        if (StringUtils.isBlank(processKey)) {
            maps = customMapper.selectRunningTasks();
        } else {
            maps = customMapper.selectRunningTasksByProcessKey(processKey);
        }
        List<RunningTask> tasks = packageTasks(maps);
        return tasks;
    }
};
List<RunningTask> tasks = managementService.executeCustomSql(customSqlExecution);
```

13.5.1 定义Mapper接口

使用MyBatis查询数据需要我们定义一个Mapper接口，例如代码清单13-10中的TaskQueryMapper接口，在接口中用@Select注解声明方法执行时需要查询的SQL，如代码清单13-11列出了TaskQueryMapper的SQL定义。

代码清单13-11 TaskQueryMapper接口定义

```
public interface TaskQueryMapper {
    @Select("select art.ID, art.NAME, art.ASSIGNEE, art.CREATE_TIME, " +
        " art.PROC_INST_ID, art.PROC_DEF_ID, arp.NAME, PROCESS_NAME " +
        " from ACT_RU_TASK art " +
        " inner join ACT_RE_PROCDEF arp on art.PROC_DEF_ID = arp.ID " +
        " where arp.SUSPENSION_STATE = '1'")
    @Results(value = {
        @Result(id = true, property = "id", column = "ID",
            javaType = String.class, jdbcType = JdbcType.VARCHAR),
        // 省略了其他字段
    })
    List<RunningTask> selectRunningTasks();
}
```

从代码清单13-11中的SQL可以看出，任务表（ACT_RU_TASK）与流程定义表（ACT_RE_PROCDEF）使用了内关联，查询的结果集中既包含任务表的字段又包含流程定义表的字段，用MyBatis的@Result注解声明了查询结果中字段与RunningTask对象属性的映射关系，最终会返回一个RunningTask集合。

如果不封装成对象也可以直接返回一个Map类型的集合，例如下面的配置：

```
List<Map<String, Object>> selectRunningTasks();
```

熟悉MyBatis的读者还可以再使用其他注解查询结果，例如在SQL中使用MyBatis的判断语句（if）等。

RunningTask对象的属性见下面的代码：

```
public class RunningTask {
    private String id;
    private String name;
    private String assignee;
    private String processName;
    private String processInstanceId;
    private String processDefinitionId;
    private Date createTime;
    // 省略getter和setter
}
```

除了自定义Bean对象外还可以使用引擎的内部实体对象，例如下面的代码返回一个TaskEntity（引擎内部类org.activiti.engine.impl.persistence.entity.TaskEntity）对象集合：

```
@Select("select * from ACT_RU_TASK RES" +
        " inner join ACT_RU_VARIABLE VAR on
        VAR.PROC_INST_ID = RES.PROC_INST_ID " +
        " where VAR.NAME = #{variableName}")
List<TaskEntity> findTasks(String variableName);
```

13.5.2 展示数据

准备好了Mapper对象以及查询结果Bean对象RunningTask，现在我们需要把结果展示出来，相关代码见代码清单13-12所示，对应的JSP文件位于chapter13/src/main/resources/WEB-INF/views/chapter13/running-tasks.jsp。

代码清单13-12 MapperQueryController控制器

```
public class MapperQueryController {
    @RequestMapping(value = "task/running")
    public ModelAndView list(
        @RequestParam(value = "processKey", required = false) final String processKey) {
        ModelAndView mav = new ModelAndView("chapter13/running-tasks");
        // 此处引用代码清单13-11
        // 省略了把结果对象封装到ModelAndView的相关代码
    }
}
```

13.6 本章小结

本章的重点只有一个——查询，从分析引擎的查询API开始，然后分别讲解了如何使用标准查询、Native查询以及基于MyBatis完全自定义的查询方式。

三种类型的查询有着各自的特点，标准查询允许以链式编程的方式查询数据；Native查询可以自由地拼接SQL语句组合查询，但是返回的结果类型都是引擎预先映射到MyBatis的；CustomSql查询最灵活，不仅SQL语句自己拼接，并且查询的结果类型也可以自定义。

第14章 管理员特性

几乎在每个系统中都会有一个监管整个系统的后台模块，根据功能的不同分配给超级管理员或者普通管理员。在工作流驱动的业务系统中管理员特性一般包含如下几个特性，本章的内容也围绕这几个特性——介绍。

- 用户与组的管理：一般的系统中都配有统一的用户模块，本章仅讨论与Activiti相关的内容。
- 流程状态：包括流程定义、流程实例、任务。
- 作业管理：可以监管定时或异步的作业任务。
- 数据库：可以直接通过系统查看数据库的记录。

14.1 流程状态

假设这样一个场景：一个正在运行的流程实例由于某些原因要暂停，从技术上来说就是“挂起”，当条件满足之后再恢复状态继续运行该流程实例。

在Activiti中，这一过程有两个对应的操作：挂起（suspend）、激活（active）。除了控制流程实例外还可以控制流程定义（ProcessDefinition）的状态（挂起或激活），并且可以选择是否激活与流程定义相关的流程实例，以及定时激活或挂起。

14.1.1 流程定义状态

在第5章中介绍了如何读取流程定义列表以及如何删除流程定义，目前的情况是启动流程需要先从“流程定义”页面选择一个流程，并且还执行删除操作。很明显这个“不妥”的设计需要改进，流程的部署以及删除属于管理员的权限范围之内，所以这两个操作对普通用户是不可见的。

为此再添加一个页面供普通用户使用，只有一个“启动”操作，并把原来的“流程定义”更名为“流程定义管理”，开放流程部署及删除功能，图14-1展示了新的菜单项。



图14-1 改进的流程列表菜单

新添加的“流程列表”菜单项的链接如下：

```
<li><a href="#" rel='chapter5/process-list?readonly=true'>
流程列表</a></li>
```

添加一个参数“readonly”并设置为“true”，然后页面会根据该参数控制流程部署、删除功能是否可见（这里为了演示简化安全控制，在实际使用中可以使用安全框架辅助处理）。

图14-2展示了新添加的“流程列表”菜单项的页面，从图中可以看到原来的流程部署区域与删除按钮已经隐藏了，这样普通用户可操作的就只有查看流程图和启动流程。

Activiti Explorer							
🏠 首页 📄 请假（普通表单） 🔄 运行中 📁 已归档 📋 流程列表 ⚙️ 管理 👤 Kermit Miao/kermit							
流程定义ID	部署ID	流程定义名称	流程定义KEY	流程描述	版本号	图片资源名称	操作
leave-countersign:1:19	1	请假流程-会签	leave-countersign	请假流程演示-会签	1	leave-countersign.png	▶ 启动
leave-mail-timeout:1:18	1	请假流程-邮件任务-销假超时提醒	leave-mail-timeout	请假流程-邮件任务-销假超时提醒	1	leave-mail-timeout.png	▶ 启动

图14-2 流程列表界面

在“流程定义管理”界面中则显示如图14-3所示的操作按钮，在该界面可以删除和挂起流程定义（挂起后显示“激活”操作按钮）。

在点击图14-3中的“挂起”按钮后会弹出如图14-4所示的对话框，可以选择现在挂起还是定时挂起，“同时挂起所有与流程定义相关的流程实例？”的含义是：如果要挂起的流程定义已经有正在运行的流程实例，则同时级联执行挂起操作。

图片资源名称	操作
leave-countersign.png	操作 ▼ 删除 挂起
leave-mail-timeout.png	

图14-3 流程定义管理界面的操作按钮

挂起流程定义

何时执行: ☒ 现在

☐ 定时: 前

☒ 同时挂起所有与流程定义相关的流程实例?

关闭

确定挂起

图14-4 挂起流程定义对话框

定时挂起是在某个特定的时间挂起, 利用引擎内置的作业 (Job) 模块来实现。在引擎配置中设置如下参数即可开启Job功能。

```
<property name="jobExecutorActivate" value="true" />
```

开启了Job功能之后引擎默认以5秒为间隔轮询检查是否有需要处理的Job, 如果作业的日期小于等于系统当前时间, 则执行该作业, 详细内容会在14.4节介绍。

在单击图14-4中的“确认挂起”按钮后提交表单, 执行代码清单14-1的代码, 根据状态的不同调用RepositoryService接口的不同方法。

代码清单14-1 更改流程定义状态

```
@Controller
@RequestMapping(value = "/chapter14/process")
public class ProcessManagerController {
    @RequestMapping(value = "{state}", method = RequestMethod.POST)
    public String changeState(@PathVariable(value = "state") String state,
        @RequestParam(value = "processDefinitionId") String processDefinitionId,
        @RequestParam(value = "cascade", required = false) boolean cascadeProcessInstances,
        @RequestParam(value = "effectiveDate", required = false) Date effectiveDate) {
        if (StringUtils.equals("active", state)) { // 激活
            repositoryService.activateProcessDefinitionById(processDefinitionId, #1
                cascadeProcessInstances, effectiveDate);
        } else if (StringUtils.equals("suspend", state)) { // 挂起
            repositoryService.suspendProcessDefinitionById(processDefinitionId, #2
                cascadeProcessInstances, effectiveDate);
        }
        return "redirect:/chapter5/process-list";
    }
}
```

代码清单14-1中的#1与#2处分别处理了激活与挂起操作, 两个方法有相同的参数, 第一个参数为流程定义ID, 第二个参数对应图14-4中的“同时挂起所有与流程定义相关的流程实例?”选项, 最后一个参数表示执行操作的日期, 如果设置为null, 表示现在挂起, 否则创建一个Job定时执行。

除了通过代码清单14-1的方式激活、挂起流程定义之外, 还可以通过流程的 (流程文件中process标签的id属性, 即图14-2中的“流程定义KEY”) ID批量更改状态, 相关API如图14-5所示。

图14-6展示了改进的流程定义列表, 其中“状态”列显示了该流程定义对象的状态 (挂起、正常)。如果一个流程定义被挂起, 此流程定义应该对普通用户不可见, 所以需要对应图14-2的列表数据过滤, 即仅查询处于激活状态的数据。

activateProcessDefinitionById(String processDefinitionId)
Activates the process definition with the given id.

activateProcessDefinitionById(String processDefinitionId, boolean activateProcessInstances, Date activationDate)
Activates the process definition with the given id.

activateProcessDefinitionByKey(String processDefinitionKey)
Activates the process definition with the given key (=id in the bpmn20.xml file).

activateProcessDefinitionByKey(String processDefinitionKey, boolean activateProcessInstances, Date activationDate)
Activates the process definition with the given key (=id in the bpmn20.xml file).

图14-5 激活流程定义的API

流程定义ID	部署ID	流程定义名称	流程定义KEY	流程描述	版本号	XML资源名称	图片资源名称	状态	操作
leave-countersign:1:930	901	请假流程-会签	leave-countersign	请假流程演示-会签	1	leave-countersign.bpmn	leave-countersign.png	挂起	操作 ▾
leave-formkey:1:925	901	请假流程-外置表单	leave-formkey	《Activiti实战》第6章的例子--外置表单	1	chapter6/leave-formkey/leave-formkey.bpmn	chapter6/leave-formkey/leave-formkey.png	挂起	操作 ▾
leave-mail-timeout:1:929	901	请假流程-邮件任务-销假超时提醒	leave-mail-timeout	请假流程-邮件任务-销假超时提醒	1	leave-mail-timeout.bpmn	leave-mail-timeout.png	正常	操作 ▾
leave-mail:1:924	901	请假流程-邮件任务	leave-mail	请假流程演示-邮件任务	1	leave-mail.bpmn	leave-mail.png	正常	操作 ▾
leave:1:928	901	请假流程-普通表单	leave	请假流程演示	1	leave.bpmn	leave.png	正常	操作 ▾

共8条数据

<<<12>>>

图14-6 包含挂起与正常状态的流程定义列表

下面的代码在查询流程定义时根据readonly的值进行了过滤，当然也可以只查询挂起的流程定义对象，刷新图14-2页面后可以看到图14-7所示的界面，比图14-6的数据少了两条。

```
processDefinitionQuery.active();
// processDefinitionQuery.suspended() 只查询挂起的数据
```

流程定义ID	部署ID	流程定义名称	流程定义KEY	流程描述
leave-mail-timeout:1:929	901	请假流程-邮件任务-销假超时提醒	leave-mail-timeout	请假流程-邮件任务-销假超时提醒
leave-mail:1:924	901	请假流程-邮件任务	leave-mail	请假流程演示-邮件任务
leave:1:928	901	请假流程-普通表单	leave	请假流程演示
payment:1:926	901	通用付款流程	payment	
purchase-callactivity:1:923	901	办公用品采购-callactivity方式	purchase-callactivity	

共6条数据

<<<12>>>

图14-7 过滤了挂起的流程定义数据

14.1.2 作业查询

上一小节中提到了定时执行挂起、激活流程定义的功能，本小节将简单介绍一下如何查询作业，在14.4节中再介绍其他与作业相关的功能。

图14-8展示了“激活流程定义”的对话框，单击“确定激活”后调用引擎接口添加一个定时作业，通过引擎提供的接口可以查询到如图14-9所示的数据。

激活流程定义

何时执行: ☐ 现在

☒ 定时: 2013-04-07 06:02:49 

☒ 同时挂起所有与流程定义相关的流程实例?

关闭

确定激活

图14-8 激活流程定义对话框

作业管理列表

作业ID	作业类型	预定时间	可重试次数	流程定义ID	流程实例ID	执行ID	异常消息	作业配置信息	操作
12601	挂起流程定义	2014-04-16 09:25:56	3	leave-formkey:1:10125				{"includeProcessInstances":true}	操作 ▾
12606	中间定时	2014-04-16 09:26:13	3	jobExecuteFail:1:10134	12602	12605		timerintermediatecatchevent1	操作 ▾

共2条数据 << < 1 > >>

图14-9 作业 (Job) 管理列表

图14-9的列表列出了Job对象的几个主要属性，根据“作业类型”的不同，某些字段的值也不尽相同，第一条记录类型为“激活流程定义”，该作业只记录需要启动的流程定义ID即可，所以“流程实例ID”、“执行ID”的值为空，只有运行时流程产生的作业才会有这两个值；第二条记录的“作业类型”为“中间定时”，我们知道中间定时事件必须由运行中的流程触发，所以流程实例ID与执行ID均有值。

最后一个属性“作业配置信息”是执行任务时的参数（纯文本类型），例如第一条记录中的“includeProcessInstances: true”用JSON格式保存表示在执行激活流程定义时同时激活流程相关的实例；而第二条记录的“timerintermediatecatchevent1”是流程定义中中间定时事件的ID。

下面的代码列出了查询作业列表的代码片段，完整的代码请参考本书配套资源中本章源码的JobController.java和job-list.jsp文件。

```
JobQuery jobQuery = managementService.createJobQuery();
List<Job> jobList = jobQuery.list();
```

下面的代码列出了展示层显示属性值的代码：

```
<td>${job.id}</td>
<td>${JOB_TYPES[job.jobHandlerType]}</td>
<td><fmt:formatDate value="${job.duedate}"
    pattern="yyyy-MM-dd hh:mm:ss"/></td>
<td>${job.retries}</td>
<td>${job.processDefinitionId}</td>
<td>${job.processInstanceId}</td>
<td>${job.executionId}</td>
<td>${job.exceptionMessage}</td>
<td>${job.jobHandlerConfiguration}</td>
```

对于“作业类型”属性，Job对象显示的是英文。下面的代码列出了目前引擎中与作业类型有关的中英文对照。

```
JOB_TYPES.put("activate-processdefinition", "激活流程定义");
JOB_TYPES.put("timer-intermediate-transition", "中间定时");
JOB_TYPES.put("timer-transition", "边界定时");
JOB_TYPES.put("timer-start-event", "定时启动流程");
JOB_TYPES.put("suspend-processdefinition", "挂起流程定义");
```

14.1.3 流程实例状态

在挂起、激活流程定义时可以选择同时级联更改相关流程实例的状态，当然也可以单独对流程实例进行操作，两者操作的API类似，只不过流程实例不能像流程定义那样可以定时执行挂起、激活操作。图14-10中的流程实例列表中列出了流程实例，并且包含了正常和挂起两种状态的数据。

通过调用下面两个RuntimeService接口的方法可以分别激活、挂起流程实例，这两个方法均有一个流程实例ID参数。

void activateProcessInstanceId(String processInstanceId)
void suspendProcessInstanceId(String processInstanceId)

Explorer首页请假（普通表单）运行中已归档流程列表管理Kermit Miao/kermit

流程定义管理
作业管理
流程实例

流程实例ID	流程定义ID	流程名称	流程版本	业务KEY	状态
1002	leave-mail-timeout:1:929	请假流程-邮件任务-销假超时提醒	1		正常 挂起
1106	timerStartEvent:1:1104	TimerStartEvent	1		正常 挂起
1115	timerInMiddle:1:1114	timerInMiddle	1		挂起 激活
1201	leave:1:928	请假流程-普通表单	1	1	挂起 激活

共4条数据<<<1>>>

图14-10 流程实例列表

图14-10中的列表查询之前已经介绍过，所以不再重复，读者可参考本书配套资源的ProcessInstanceManagerController.java和processinstance-list.jsp。

下面的代码展示了显示图14-10中处理“状态”列的逻辑处理，根据流程实例的状态显示对应的按钮。

```
<c:forEach items="${page.result }" var="pi">
<td>
    ${pi.suspended ? '挂起' : '正常'}
    <c:if test="${pi.suspended}">
        <a href="${ctx }/chapter14/processinstance
            /active/${pi.id}">激活</a>
    </c:if>
    <c:if test="${!pi.suspended}">
        <a href="${ctx }/chapter14/processinstance
            /suspend/${pi.id}">挂起</a>
    </c:if>
</td>
</c:forEach>
```

代码清单14-2列出了处理挂起、激活操作的Java代码，操作成功后返回流程实例列表。

代码清单14-2 更改流程实例状态

```
@RequestMapping(value = "${state}/{processInstanceId}")
public String changeState(@PathVariable("state") String state,
    @PathVariable("processInstanceId") String processInstanceId) {
    if (StringUtils.equals("active", state)) {
        runtimeService.activateProcessInstanceId(processInstanceId);
    } else {
        runtimeService.suspendProcessInstanceId(processInstanceId);
    }
    return "redirect:/chapter14/processinstance/list";
}
```

需要注意的是，激活与挂起流程实例操作默认会级联操作与该实例相关的活动，所以对于“任务列表”、“参与的流程”，需要根据流程的状态过滤，也就是只查询非挂起的数据。

查看ProcessInstanceQuery与TaskQuery对象均提供了一个“active()”方法，调用此方法后只能查询到正常状态的数据，例如以下两行代码分别可以查询正常状态的流程实例及任务。

```
runtimeService.createProcessInstanceQuery().active().list()
taskService.createTaskQuery().active().list()
```

“请假（普通表单）”的“任务列表”使用的标准查询，只要在调用list()方法之前调用一次active()即可，修改后的代码如下：

```
List<Task> todoList = taskService.createTaskQuery()
    .processDefinitionKey("leave").taskAssignee(userId)
    .active().list();
// 根据当前人未签收的任务
List<Task> unsignedTasks = taskService.createTaskQuery()
    .processDefinitionKey("leave").taskCandidateUser(userId)
    .active().list();
```

在上一章中，为了满足复杂查询使用了NativeQuery查询方式，标准查询的过滤方式就不能满足需求了，所以只能在SQL语句中添加过滤条件只查询正常状态的数据。

在原有的查询SQL的where关键字后面添加如下的条件即可，字段“SUSPENSION_STATE_”值为“1”表示正常状态，为“2”表示挂起状态。

```
SUSPENSION_STATE_ = '1'
```

读者可以参考本书配套资源中本章源码的TaskController.java与ExecutionController.java文件的SQL语句。

14.2 作业管理

一般来说，管理员除了能查看等待执行的作业之外，还可以删除作业和手动触发作业的执行动作。ManagementService接口的两个方法分别实现作业的删除与执行功能，API如下：

```
deleteJob(String jobId);
executeJob(String jobId)
```

在图14-9对应的代码的基础上再添加下面的操作链接，对应的控制层代码如代码清单14-3所示。

```
<td>
  <a href="/execute/{job.id}">执行</a>
  <a href="/delete/{job.id}">删除</a>
</td>
```

代码清单14-3 操作作业的控制层代码

```
// 删除作业
@RequestMapping(value = "delete/{jobId}")
public String deleteJob(@PathVariable("jobId") String jobId) {
    managementService.deleteJob(jobId);
    return "redirect:/chapter14/job/list";
}

// 手动执行作业
@RequestMapping(value = "execute/{jobId}")
public String executeJob(@PathVariable("jobId") String jobId) {
    managementService.executeJob(jobId);
    return "redirect:/chapter14/job/list";
}
```

14.2.1 作业执行原理

在所接触的项目中有任务调度模块可以代替人工定时处理一些系统任务，例如定期备份、定期导出报表等任务。

Activiti中的作业模块的实现原理与大多数任务调度框架类似，利用定时刷新（间隔5秒）的方式查询需要执行的作业，具体的执行过程可以参考如图14-11所示的作业执行原理图。

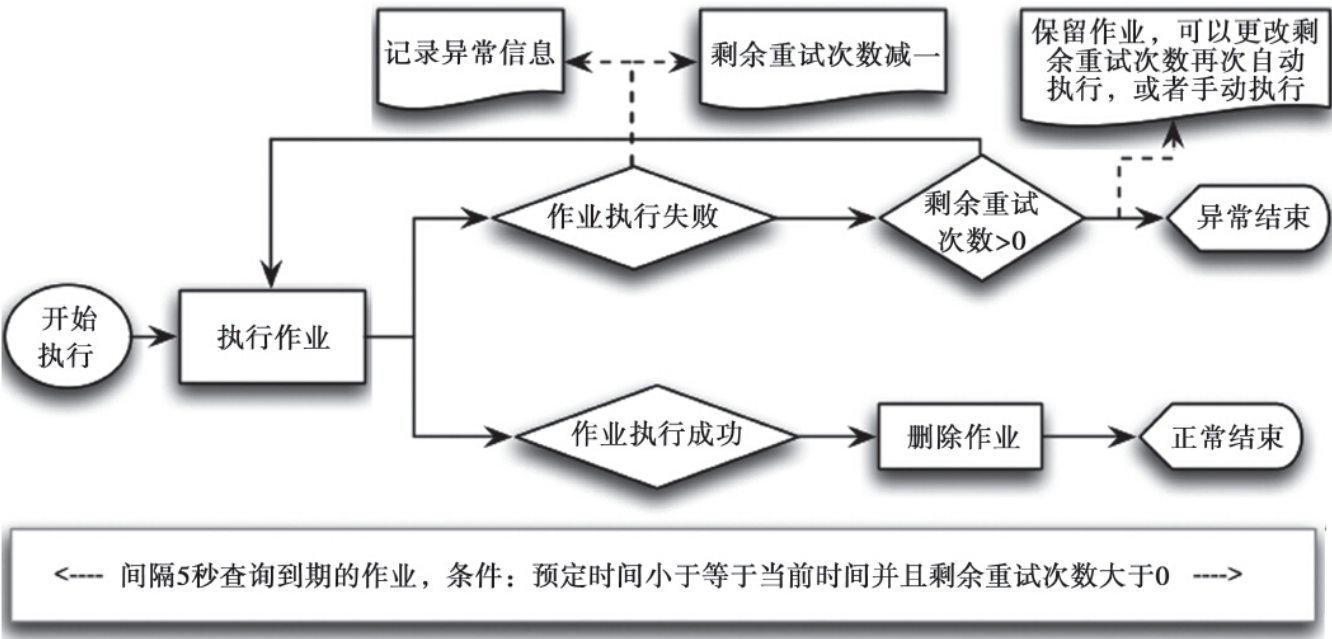


图14-11 Activiti中作业执行原理图

图14-11中底部的长框表示查询到期作业的守护线程，根据框中的条件进行过滤。

图14-11显示作业执行后根据执行结果分为两种不同的处理方式：如果执行成功，处理的逻辑很简单，直接把作业记录删除就完成了；如果在作业执行失败后需要处理异常，不仅要记录执行的异常信息，还要重新执行作业（同时可重试次数减一），以此条件循环，直到作业执行成功后正常结束或剩余重试次数为0后结束执行，若重试了3次后作业还未能正常执行，此时可以选择手动执行（不限制次数）。

14.2.2 作业执行异常

在作业执行过程中可能会遇到业务异常或者系统异常，对于这种情况可以查询作业执行时的异常堆栈（Exception Stacktrace）信息来分析失败的原因。

为此我们用一个简单的例子模拟一下作业执行失败的情况，做法是在执行一个Java Service任务时故意抛出一个RuntimeException异常，判断是否抛出异常的条件是当前流程实例相关作业的重试次数是否大于0，因为一旦将作业执行失败重试次数设置为0，就可以手动执行作业。

图14-12展示了一个简单的流程，该流程启动后会创建一个定时作业，10秒后继续执行，此时查看图14-13的“作业管理”列表的数据，列“异常消息”显示“本次作业执行失败，再次执行可以成功！”。该异常信息是由代码清单14-4中的Java类抛出的，观察控制台的输出会发现异常消息输出了三次。

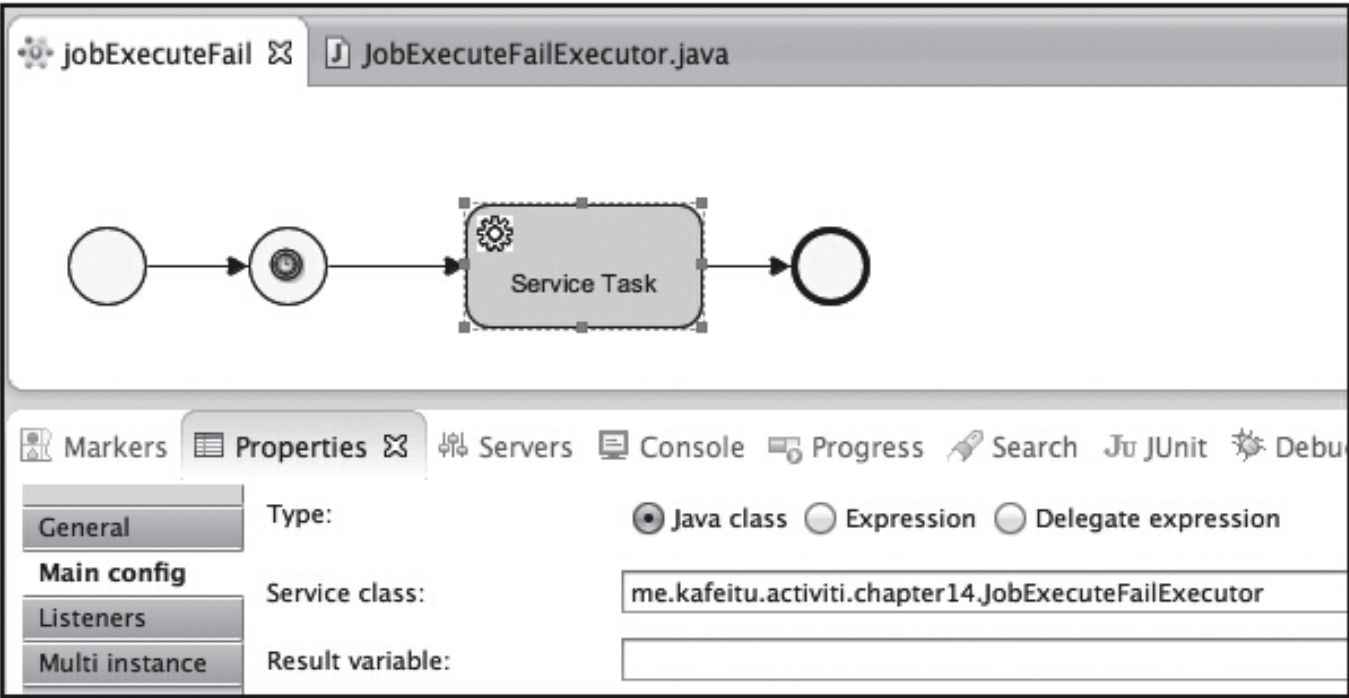


图14-12 定时作业自动执行失败示例流程

作业ID	作业类型	预定时间	可重试次数	流程定义ID	流程实例ID	执行ID	异常消息	作业配置信息
1559	中间定时	2013-04-16 09:29:03	0		1555	1558	本次作业执行失败，再次执行可以成功！	timerintermediatecatchevent1

图14-13 启动图14-12的流程10秒后作业的执行状态

代码清单14-4列出了图14-12中与“Service Task”的“Service class”属性关联的Java类。

代码清单14-4 图14-12中“Java Service”实现类

```
public class JobExecuteFailExecutor implements JavaDelegate {
    public void execute(DelegateExecution execution) throws Exception {
        ManagementService managementService = execution
            .getEngineServices().getManagementService();
        Job job = managementService.createJobQuery()
            .processInstanceId(execution.getProcessInstanceId()).singleResult();
        if (job.getRetries() > 0) {
            throw new RuntimeException("本次作业执行失败，再次执行可以成功！");
        }
    }
}
```

对于图14-11作业执行失败的情况，引擎会记录异常信息（图14-13中的“异常消息”）以及异常堆栈（见图14-14所示），也就是引擎通过tryhttp://www.hzcourse.com/resource/readBook?path=/openresources/teach_ebook/uncompressed/15030/OEBPS/Text/...catch语句捕获异常堆栈并保存到数据库。

将鼠标移动到“异常消息”列后会显示图14-14中的异常堆栈，通过引擎提供的接口可以根据作业ID读取异常堆栈信息。

```
Map<String, String> exceptionStacktraces =
    new HashMap<String, String>();
for (Job job : jobList) {
    if (StringUtil.isNotBlank(job.getExceptionMessage())) {
        exceptionStacktraces.put(job.getId(), managementService
            .getJobExceptionStacktrace(job.getId()));
    }
}
```

异常消息	作业配置信息	操
本次作业执行失败，再次执行可以成功！	timerintermediatecatchevent1	
本java.lang.RuntimeException: 本次作业执行失败，再次执行可以成功！ at me.kafeitu.activiti.chapter14.JobExecuteFailExecutor.execute(JobExecuteFailExecutor.java:18) at org.activiti.engine.impl.delegate.JavaDelegateInvocation.invoke(JavaDelegateInvocation.java:34) at org.activiti.engine.impl.delegate.DelegateInvocation.proceed(DelegateInvocation.java:37) at org.activiti.engine.impl.delegate.DefaultDelegateInterceptor.handleInvocation(DefaultDelegateInterceptor.java:25) at org.activiti.engine.impl.bpmn.behavior.ServiceTaskJavaDelegateActivityBehavior.execute(ServiceTaskJavaDelegateActivityBehavior.java:49) at org.activiti.engine.impl.bpmn.behavior.ServiceTaskJavaDelegateActivityBehavior.execute(ServiceTaskJavaDelegateActivityBehavior.java:40) at org.activiti.engine.impl.bpmn.helper.ClassDelegate.execute(ClassDelegate.java:116) at org.activiti.engine.impl.pvm.runtime.AtomicOperationActivityExecute.execute(AtomicOperationActivityExecute.java:44) at org.activiti.engine.impl.interceptor.CommandContext.performOperation(CommandCo		

图14-14 作业执行失败后显示的异常堆栈信息

最后把Map集合传递给展示层，根据作业ID获取异常堆栈信息即可，代码如下：

```
<td title="${exceptionStacktraces[job.id]}">
    ${job.exceptionMessage}</td>
```

假如管理员看到图14-14的异常后很快将Bug修复了，需要重新执行该作业：一种办法是通过手动方式执行作业；另外一个办法是更改作业的“可重试次数”属性以便再次被引擎检测到，这样可以再次自动触发作业执行。

14.2.3 独占与异步

或许读者已经注意到，在使用Activiti Designer设计流程时任务属性中的“Asynch-ronous”与“Exclusive”，之所以前面的章节没有介绍，是因为要理解异步与独占和作业有一定的关系，现在已经理解了作业执行的原理，是时候介绍这两个属性的用途了。

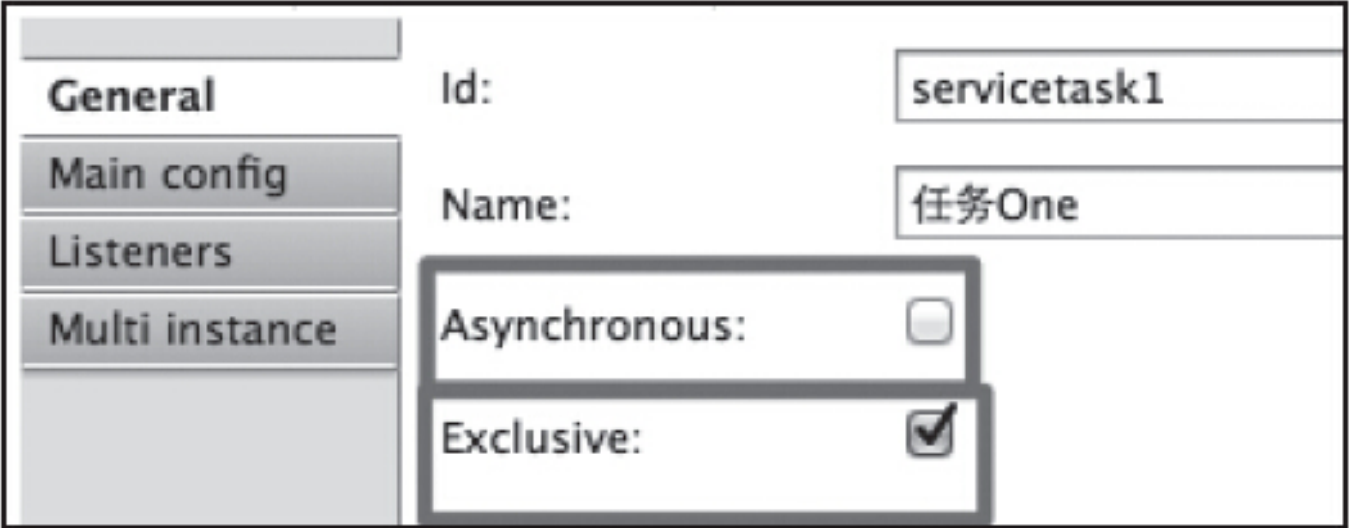


图14-15 用Activiti Designer设计流程时的异步与独占选项

异步 (Asynchronous) 是针对并行任务处理的一个概念。图14-16所示的是一个包含并行分支的流程，其中并行分支包含三个Java Service任务，在默认情况下并行分支的输出流关联的活动都是同步执行且是独占 (稍后介绍) 的，也就是说先执行“任务One”再执行“任务Two”最后执行“任务Three”。如果并行的任务执行时间比较长 (可能几分钟也可能是一个小时)，那么后面的任务也要等待前面的任务执行完成。

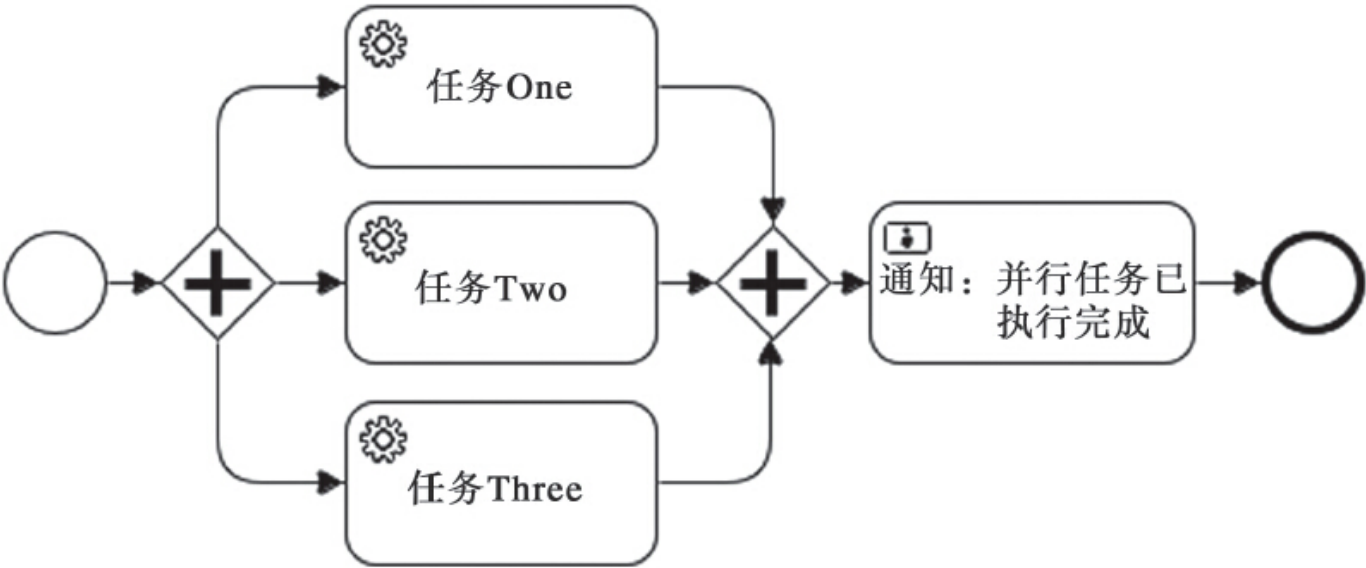


图14-16 包含并行分支的流程

如果把三个任务都设置为异步，也就是勾选图14-15的“Asynchronous”选项，在该流程启动之后会创建三个类型为“async-continuation”的作业记录，用于记录异步作业的到期时间 (与其他的定时作业的到期有所不同，稍后介绍)。

独占 (Exclusive) 是Activiti引擎为了解决乐观锁^[1]问题采用的一种解决办法，在jBPM4也采用了相同的处理方式。简单来说就是同一个流程实例在同一时刻只能执行一个任务，从而避免多个任务同时执行导致事务问题。

假如将图14-16的三个并行任务都设置为异步执行，对应的XML如下 (完整的流程定义可参考本书配套资料中本章源码的src/main/resources/chapter14/asynchronousAnd-Exclusive.bpmn文件)：

```
<serviceTask id="servicetask1" name="任务One"
  activiti:async="true" activiti:exclusive="true"
  activiti:class="me.kafeitu.activiti.chapter14
    .AsynchronousExecutor"/>
```

“activiti:async”对应“asynchronous”选项，“activiti:exclusive”对应“Exclusive”选项，其中“activiti:exclusive”可以省略，如果不设置引擎，默认使用独占模式。

为了彻底理解这两个选项对流程的影响，我们为三个Java Service任务设置了一个执行类 (实现了JavaDelegate接口)，用于打印执行日志，见代码清单14-5。

代码清单14-5 图14-16中“Java Service”实现类AsynchronousExecutor

```
public class AsynchronousExecutor implements JavaDelegate {
    private Logger logger = LoggerFactory.getLogger(getClass());
    public void execute(DelegateExecution execution) throws Exception {
        logger.info(execution.getCurrentActivityName() + "----开始执行任务");
        Long sleepSeconds = (Long) execution.getVariable("sleepSeconds");
        // 根据启动流程时表单填写的只睡眠来模拟耗时较长的情况
        Thread.sleep(sleepSeconds * 1000L);
        logger.info(execution.getCurrentActivityName() + "----任务完成");
    }
}
```


有了理论知识，再结合AsynchronousExecutor的日志输出及已经做好的作业列表，我们就可以分别根据异步、独占选项的特点来分析测试类的执行结果，以此根据实际需要决定如何使用哪种模式。

首先对图14-16的流程进行部署，然后启动流程输入每个任务等待的时长（模拟耗时较长的场景，默认为5分钟）。

启动流程 — [AsynchronousExecutor]，版本号：1

睡眠时长（秒）：

◀ 返回列表

▶ 启动流程

图14-17 启动流程并设置任务执行等待时长为10秒钟

在创建异步的任务（图14-18）时会同时创建对应的作业记录（如图14-19所示），到期时间为创建任务的时间加5分钟，图14-18中的到期时间为04:33:30，也就是说流程的启动时间为04:28:30。

图14-18中的“作业类型”为“异步锁”，英文标示见图14-19的“HANDLER_TYPE_”字段，可以直译为“异步继续”，但是笔者认为翻译成“异步锁”更为贴切。

作业ID	作业类型	预定时间	可重试次数	流程定义ID	流程实例ID	执行ID	异常消息	作业配置信息
3411	异步锁		3		3401	3408		到期时间: 2013-04-21 04:33:30 锁标示(UUID): 6ffd6df3-ba1e-446c-8ba2-84f1f087001c
3412	异步锁		3		3401	3409		到期时间: 2013-04-21 04:33:30 锁标示(UUID): 6ffd6df3-ba1e-446c-8ba2-84f1f087001c
3413	异步锁		3		3401	3410		到期时间: 2013-04-21 04:33:30 锁标示(UUID): 6ffd6df3-ba1e-446c-8ba2-84f1f087001c
共3条数据 << < 1 > >>								

图14-18 将任务设置异步且独占后创建的作业

SELECT * FROM ACT_RU_JOB;														
ID_	REV_	TYPE_	LOCK_EXP_TIME_	LOCK_OWNER_	EXCLUSIVE_	EXECUTION_ID_	PROCESS_INSTANCE_ID_	PROC_DEF_ID_	RETRIES_	EXCEPTION_STACK_ID_	EXCEPTION_MSG_	DUEDATE_	REPEAT_	HANDLER_TYPE_
3413	2	message	2013-04-21 16:33:30.124	6ff06df3-ba1e-446c-8ba2-84f1f087001c	TRUE	3410	3401	null	3	null	null	null	null	async-continuation
3412	2	message	2013-04-21 16:33:30.124	6ff06df3-ba1e-446c-8ba2-84f1f087001c	TRUE	3409	3401	null	3	null	null	null	null	async-continuation
3411	2	message	2013-04-21 16:33:30.124	6ff06df3-ba1e-446c-8ba2-84f1f087001c	TRUE	3408	3401	null	3	null	null	null	null	async-continuation
(3 rows, 5 ms)														

图14-19 ACT_RU_JOB表与图14-18的记录对应

图14-20是将任务设置异步且独占后控制台的输出结果。

2013-04-21 16:28:30,166	[pool-1-thread-1]	INFO	[me.kafeitu.activiti.chapter14.AsynchronousExecutor]	- 任务One——开始执行任务
2013-04-21 16:28:40,172	[pool-1-thread-1]	INFO	[me.kafeitu.activiti.chapter14.AsynchronousExecutor]	- 任务One——任务完成
2013-04-21 16:28:40,199	[pool-1-thread-1]	INFO	[me.kafeitu.activiti.chapter14.AsynchronousExecutor]	- 任务Two——开始执行任务
2013-04-21 16:28:50,203	[pool-1-thread-1]	INFO	[me.kafeitu.activiti.chapter14.AsynchronousExecutor]	- 任务Two——任务完成
2013-04-21 16:28:50,228	[pool-1-thread-1]	INFO	[me.kafeitu.activiti.chapter14.AsynchronousExecutor]	- 任务Three——开始执行任务
2013-04-21 16:29:00,232	[pool-1-thread-1]	INFO	[me.kafeitu.activiti.chapter14.AsynchronousExecutor]	- 任务Three——任务完成

图14-20 将任务设置异步且独占后控制台的输出

从图14-20可以看出，虽然设置了异步但是和预期的效果不同，看起来还是与默认的同步结果一致，预期的结果应该是这样的：

任务One---开始执行任务
任务Two---开始执行任务
任务Three---开始执行任务
任务One---任务完成
任务Two---任务完成
任务Three---任务完成

导致这样结果的原因在于开启了独占，因为在独占模式下在同一时刻只能执行一个任务，所以“抵消”了异步功能。好的，我们把独占模式设置为“false”再观察结果。下面的配置把任务的独占模式设置为false，这样才是真正的异步任务处理。

```
<serviceTask id="servicetask1" name="任务One"
  activiti:async="true" activiti:exclusive="false"
  activiti:class="me.kafeitu.activiti.chapter14
    .AsynchronousExecutor"/>
```

按照刚刚的步骤部署后（流程定义的版本号为2）启动流程，查看作业列表和数据库记录与上一次基本无差异，但是流程执行结果则大不相同，并且还抛出了异常，如图14-21所示。

```
2013-04-21 20:21:25,808 [pool-1-thread-2] INFO [me.kafeitu.activiti.chapter14.AsynchronousExecutor] - 任务Two---开始执行任务
2013-04-21 20:21:25,808 [pool-1-thread-1] INFO [me.kafeitu.activiti.chapter14.AsynchronousExecutor] - 任务One---开始执行任务
2013-04-21 20:21:25,808 [pool-1-thread-3] INFO [me.kafeitu.activiti.chapter14.AsynchronousExecutor] - 任务Three---开始执行任务
2013-04-21 20:21:35,821 [pool-1-thread-2] INFO [me.kafeitu.activiti.chapter14.AsynchronousExecutor] - 任务Two---任务完成
2013-04-21 20:21:35,821 [pool-1-thread-3] INFO [me.kafeitu.activiti.chapter14.AsynchronousExecutor] - 任务Three---任务完成
2013-04-21 20:21:35,821 [pool-1-thread-1] INFO [me.kafeitu.activiti.chapter14.AsynchronousExecutor] - 任务One---任务完成
Exception in thread "pool-1-thread-1" org.activiti.engine.ActivitiOptimisticLockingException: ExecutionEntity[3505] was
updated by another transaction concurrently
    at org.activiti.engine.impl.db.DbSqlSession.flushUpdates(DbSqlSession.java:653)
    at org.activiti.engine.impl.db.DbSqlSession.flush(DbSqlSession.java:461)
    at org.activiti.engine.impl.interceptor.CommandContext.flushSessions(CommandContext.java:168)
    at org.activiti.engine.impl.interceptor.CommandContext.close(CommandContext.java:115)
    at org.activiti.engine.impl.interceptor.CommandContextInterceptor.execute(CommandContextInterceptor.java:70)
    at org.activiti.spring.SpringTransactionInterceptor$1.doInTransaction(SpringTransactionInterceptor.java:42)
    at org.springframework.transaction.support.TransactionTemplate.execute(TransactionTemplate.java:130)
    at org.activiti.spring.SpringTransactionInterceptor.execute(SpringTransactionInterceptor.java:40)
    at org.activiti.engine.impl.interceptor.LogInterceptor.execute(LogInterceptor.java:31)
    at org.activiti.engine.impl.jobexecutor.ExecuteJobsRunnable.run(ExecuteJobsRunnable.java:46)
    at java.util.concurrent.ThreadPoolExecutor$Worker.runTask(ThreadPoolExecutor.java:895)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:918)
    at java.lang.Thread.run(Thread.java:680)
Exception in thread "pool-1-thread-2" org.activiti.engine.ActivitiOptimisticLockingException: ExecutionEntity[3505] was
updated by another transaction concurrently
    ...
    at java.lang.Thread.run(Thread.java:680)
```

图14-21 更改任务属性为异步非独占后的执行结果

从图14-21可以看出，任务的执行顺序的确是随机的，不再按照One、Two、Three的顺序，也就是说引擎为这三个任务分别开启了一个线程（Thread）单独执行，而在独占模式下同时只有一个线程运行，但是最终的结果是引擎抛出了乐观锁异常（ActivitiOptimisticLockingException）。从图14-21中可以看出异常被抛出了两次（thread-1、thread-2），原因是第一个任务完成时更改了流程状态信息，接着其他两个任务完成也要更改，但是引擎判断发现已经被更改过了，所以抛出了乐观锁异常。

如何解决这个问题呢？既然引擎不能支持真正的异步执行，可以考虑从业务角度去找解决办法，可以把多个并行任务合并成一个Java Service任务，创建三个线程分别执行原来需要执行的任务，这样就可以绕过乐观锁问题。

[1] <http://zh.wikipedia.org/wiki/乐观并发控制>

14.3 删除流程实例

在14.1节中介绍了如何控制流程的状态，可以挂起、激活流程。如果正常运行的某个流程因业务数据错误不再需要了，此时可以选择删除流程实例。删除又分为运行时与历史（归档）两种：RuntimeService接口与HistoryService分别提供了一个API用来删除对应类型的流程数据。

下面的API是RuntimeService提供的，在Javadoc中也做了说明，可以删除正常运行的流程实例，第二个参数的意思是删除流程实例的原因。

```
deleteProcessInstance(String processInstanceId,
                      String deleteReason)
Delete an existing runtime process instance.
```

HistoryService的API如下所示，无需传递删除原因，因为运行时的流程实例被删除之后会把删除原因记录在历史流程实例中，这样可以跟踪流程删除的原因，方便业务方面的跟踪。

```
deleteHistoricProcessInstance(String processInstanceId)
Deletes historic process instance.
```

图14-22展示了本章示例中添加的两个菜单项，分别为“运行中流程”与“已归档流程”，用于管理正在运行和已经结束的流程实例。

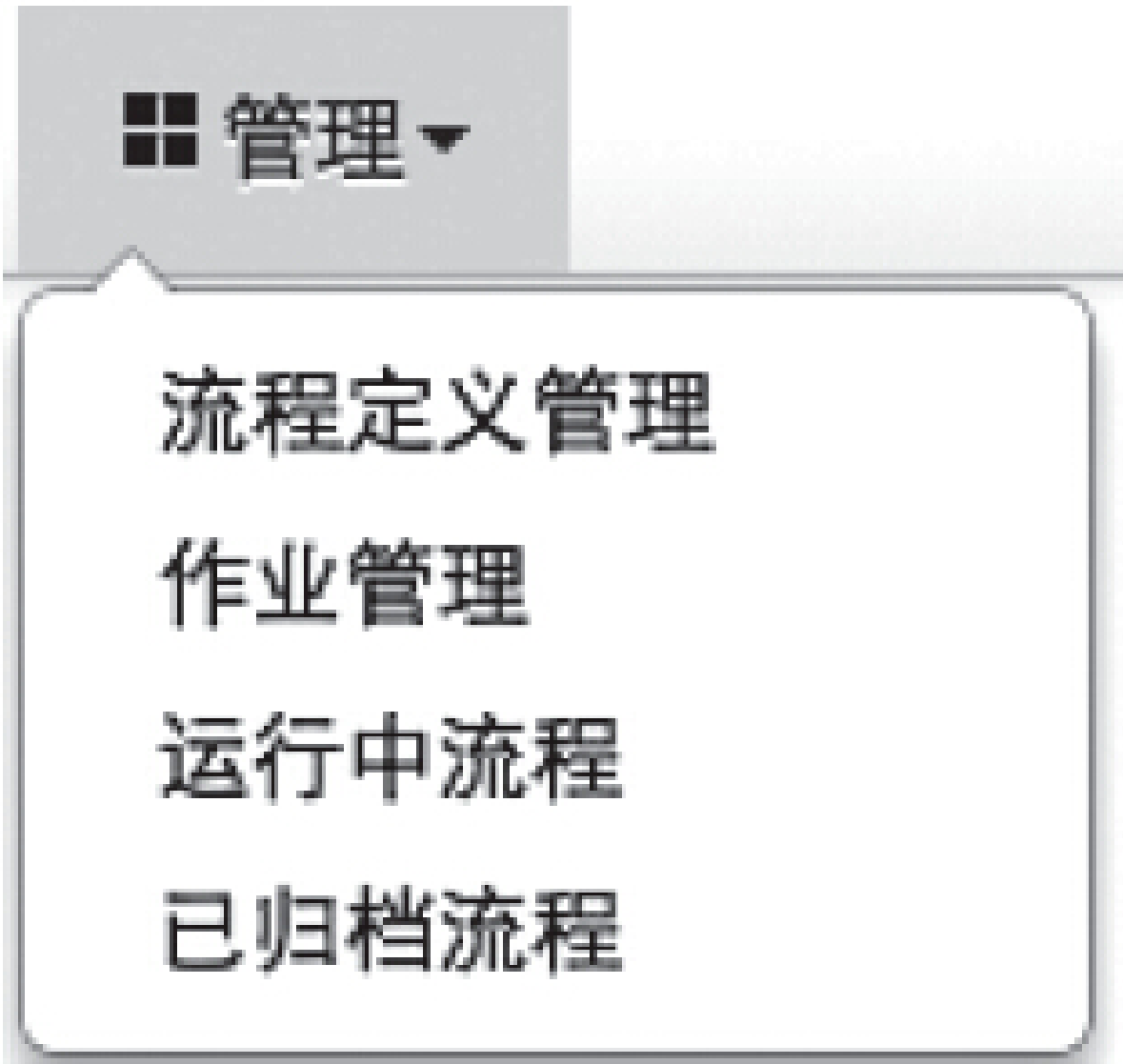


图14-22 新的管理菜单项

单击图14-22中“运行中流程”菜单项后进入如图14-23所示的流程实例列表。单击图14-23中的删除按钮后弹出如图14-24所示的对话框，提示输入删除原因，确定之后将请求发送至/chapter14-management/chapter14/processinstance/delete/13223。

流程实例ID	流程定义ID	流程名称	流程版本	业务KEY	状态	操作
1002	leave-mail-timeout:1:929	请假流程-邮件任务-销假超时提醒	1		挂起	✓激活 ✕删除
1106	timerStartEvent:1:1104	TimerStartEvent	1		正常	🔒挂起 ✕删除
1115	timerInMiddle:1:1114	timerInMiddle	1		正常	🔒挂起 ✕删除
1201	leave:1:928	请假流程-普通表单	1	1	挂起	✓激活 ✕删除
1208	purchase-callactivity:1:923	办公用品采购--callactivity方式	1		正常	🔒挂起 ✕删除

共14条数据<<<123>>>

图14-23 添加了删除按钮的运行中流程实例列表

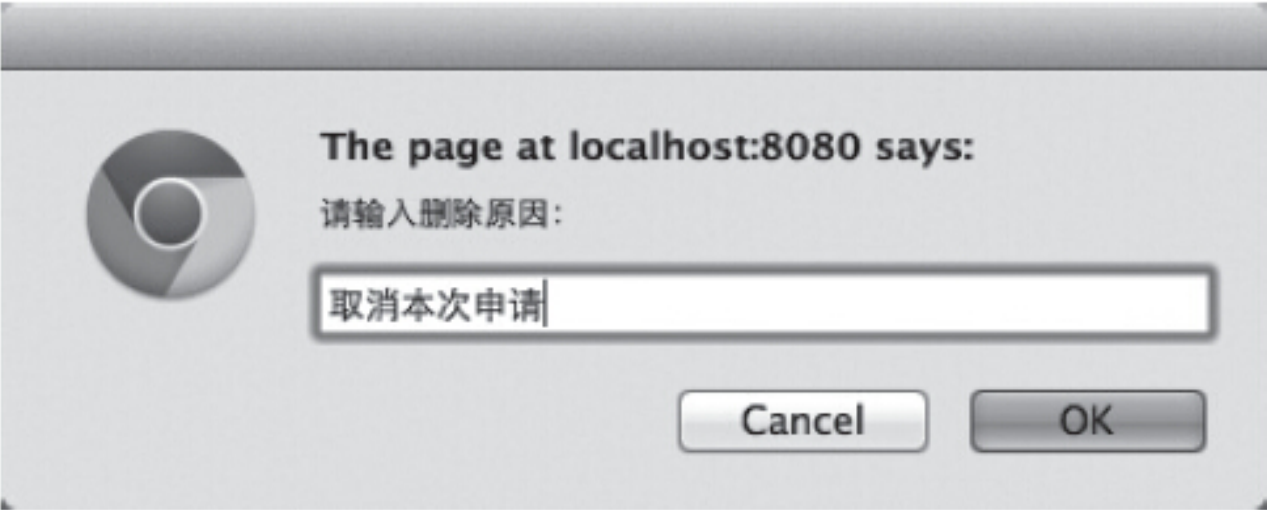


图14-24 单击图14-23中的删除按钮后提示输入删除原因

请求中的13223是流程实例的ID。控制器的代码如下所示（ProcessInstanceManagerController.java），由于展示层发送的是Ajax请求所以控制器返回一个true表示成功执行，同时该流程也被标记为已结束，可以在已归档流程中查看结束原因，如图14-25中的第2、3条记录的“结束原因”列所示。

```
@RequestMapping(value = "delete/{processInstanceId}")
@ResponseBody
public boolean delete (@PathVariable("processInstanceId")
    String processInstanceId,
    @RequestParam("deleteReason") String deleteReason) {
    runtimeService.deleteProcessInstance(processInstanceId,
        deleteReason);
    return true;
}
```

流程实例ID	所属流程	流程定义ID	启动时间	流程启动人	结束时间	父流程ID	结束原因	操作
1002	请假流程-邮件任务-销假超时提醒	leave-mail-timeout:1:929	2013-04-06 08:33:06	kermit	2013-04-21 11:14:22		null	删除
1106	TimerStartEvent	timerStartEvent:1:1104	2013-04-06 08:44:59		2013-04-21 11:37:52		该流程重复	删除
1311	通用付款流程	payment:1:926	2013-04-08 08:38:18	thomas	2013-04-21 11:36:19	1208	取消本次申请	删除
1543	该作业在执行时会抛出错误	jobExecuteFail2:1518	2013-04-16 09:28:22	kermit	2013-04-16 09:28:46			删除
3545	AsynchronousExecutor	asynchronousExecutor:1:3208	2013-04-21 08:44:38	kermit	2013-04-21 09:59:31		sdf	删除

共5条数据 << < 1 > >>

图14-25 已归档流程实例列表

在第13章的HistoryProcessInstanceController.java控制器中添加如下代码完成删除已归档流程实例的功能，成功后返回列表。

```
@RequestMapping(value = "finished/delete/{pid}")
public String deleteProcessInstance(@PathVariable("pid")
    String processInstanceId,
    RedirectAttributes redirectAttributes) {
    redirectAttributes.addFlashAttribute("message",
        "ID为" + pid + "的历史流程已删除！");
    historyService
        .deleteHistoricProcessInstance(processInstanceId);
    return "redirect:/chapter13/history/process"
        + "/finished/manager";
}
```

14.4 流程定义权限控制

要控制流程中某些用户任务能够被哪些人或组签收，可以通过为该任务设置候选人或候选组，如果把需求再提升一级，要控制哪些候选人或候选组可以启动哪些流程，该如何实现呢？

14.4.1 权限拦截

从Activiti 5.10版本开始引擎把用户任务的候选配置移植到了流程级别上，从而可以根据登录系统的用户所拥有的角色决定是否有权限查询、启动流程。图14-26是办公用品采购流程（子流程版本）的一个副本，把ID改为了“purchase-authority”表示一个特别的版本添加了候选组或候选人限制，即图中的“Candidate start users”与“Candidate start groups”属性。

Process	Id:	purchase-authority
Listeners	Name:	办公用品采购--添加了启动流程权限
	Namespace:	http://www.kafeitu.me/activiti
	Candidate start users (comma separated):	kermit
	Candidate start groups (comma separated):	supportCrew
	Documentation:	用候选人或候选组控制流程启动权限

图14-26 为流程添加候选人和候选组

对应的XML如下：

```
<process id="purchase-authority" isExecutable="true"
name="办公用品采购--添加了启动流程权限"
activiti:candidateStarterUsers="kermit"
activiti:candidateStarterGroups="supportCrew">
```

设置了属性就需要在查询流程定义对象时进行过滤，可以通过下面的方法过滤指定用户可以启动的流程。

```
repositoryService.createProcessDefinitionQuery()
.startableByUser(userId).list();
```

如果使用上面的代码过滤流程定义对象，得到的结果（用户thomas属于supportCrew组）只有一条记录，原来已经部署的流程和没有设置候选人或候选组的流程定义对象将不会包含在结果中。很显然这不是我们期望的结果，我们希望用户能查询到未设置候选属性及设置了候选属性且用户有权查看的流程定义数据。

很遗憾，目前引擎没有直接提供类似的接口，也没有提供ProcessDefinition的Native Query查询API，所以我们要想一些“歪招”来实现。RepositoryService接口中提供了一个这样的方法（如图14-27所示），利用这个方法可以在流程启动时读取流程定义的相关候选属性，然后与当前用户的属性（ID或者所属组）进行校验，以此来决定用户是否可以启动流程。

List<IdentityLink>

getIdentityLinksForProcessDefinition(String processDefinitionId)
Retrieves the IdentityLinks associated with the given process definition.

图14-27 RepositoryService接口提供的与流程定义的候选属性有关的方法

在启动流程时需要读取启动流程的表单（无论动态表单还是外置表单），在读取表单之前进行权限拦截，更改ProcessDefinitionController.java类的readStartForm方法，见代码清单14-6所示。

代码清单14-6 获取启动流程表单时添加权限拦截

```
@RequestMapping(value = "getform/start/{processDefinitionId}")
public ModelAndView readStartForm(@PathVariable("processDefinitionId")
String processDefinitionId, HttpServletRequest request,
RedirectAttributes redirectAttributes) throws Exception {
ProcessDefinition processDefinition = repositoryService.createProcessDefinitionQuery()
.processDefinitionId(processDefinitionId).singleResult();
User user = UserUtil.getUserFromSession(request.getSession()); // 获取当前用户
List<Group> groupList = (List<Group>) request.getSession().getAttribute("groups");
boolean startable = false; // 是否可以启动流程的临时变量
if (identityLinks == null || identityLinks.isEmpty()) { // 如果没有设置表示可以启动
startable = true;
}
// 读取流程定义的候选人与候选组
List<IdentityLink> identityLinks = repositoryService
.getIdentityLinksForProcessDefinition(processDefinition.getId());
for (IdentityLink identityLink : identityLinks) {
if (StringUtil.isBlank(identityLink.getUserId()) && // 候选人
identityLink.getUserId().equals(user.getId())) {
startable = true;
break;
}
if (StringUtil.isBlank(identityLink.getGroupId())) { // 候选组
for (Group group : groupList) { // 与当前人的组循环匹配
if (group.getName().equals(identityLink.getGroupId())) {
startable = true;
break;
}
}
}
}
}
if (!startable) { // 如果不能启动设置异常信息并返回流程列表
redirectAttributes.addFlashAttribute("error",
"您无权启动【" + processDefinition.getName() + "】流程！");
return new ModelAndView("redirect:/chapter5/process-list-view");
}
http://www.hzcourse.com/resource/readBook?path=/openresources/teach_ebook/uncompressed/15030/OEBPS/Text/... // 省略了读取表单的代码
}
```

以用户“henry”登录系统，启动刚刚部署的“purchase-authority”流程后显示如图14-28所示的界面，左下方的粗线边框中显示了“您无权启动【办公用品采购--添加了启动流程权限】流程！”信息。



图14-28 启动流程时的权限拦截

14.4.2 设置候选启动人和候选启动组

除了通过在设计流程时设置候选人和候选组，还可以动态地为流程定义设置候选启动人和候选启动组。如图14-29所示，引擎在RepositoryService接口中提供了API为已经部署的流程定义动态添加候选启动人和候选启动组。

```
addCandidateStarterGroup(String processDefinitionId, String groupId)
Authorizes a candidate group for a process definition.
addCandidateStarterUser(String processDefinitionId, String userId)
Authorizes a candidate user for a process definition.
```

图14-29 RepositoryService接口中提供的添加候选启动人和候选启动组的API

对应地，也可以动态地移除候选启动人和候选启动组。引擎也提供了与图14-29对应的两个方法，如图14-30所示。

```
deleteCandidateStarterGroup(String processDefinitionId, String groupId)
Removes the authorization of a candidate group for a process definition.
deleteCandidateStarterUser(String processDefinitionId, String userId)
Removes the authorization of a candidate user for a process definition.
```

图14-30 RepositoryService接口中提供的移除候选启动人和候选启动组的API

在图14-31的流程定义记录中，“候选启动（人）”与“候选启动（组）”列的数据就是通过图14-29的API添加的，单击图14-31中的“候选启动”选项后可以看到如图14-32所示的设置对话框，勾选后单击对话框的“确定”按钮即可。

流程定义ID	部署ID	流程定义名称	流程定义KEY	流程描述	版本号	XML	流程图	状态	候选启动	操作
asynchronousExecutor:1:3208	3205	AsynchronousExecutor	asynchronousExecutor	异步与独占	1	查看	查看	正常	人[4], 组[1]	操作 ▾
候选启动（人）		候选启动（组）								
- Andy Zhao - Bill Zheng - Henry Yan - Kermit Miao		后勤人员								

删除

候选启动

挂起

图14-31 为流程定义设置了候选启动人和候选启动组



图14-32 设置候选启动（人、组）对话框

在ProcessManagerController.java类中添加了一个方法用于接收单击图14-32中“确定”按钮发送的请求，以获取前台的三个参数：流程定义ID、用户（User）数组、组（Group）数组，然后把请求转交给ProcessDefinitionService.java（见代码清单14-7）类的setStartables()方法处理，见代码清单14-8所示。

代码清单14-7 接收设置候选启动请求

```
@RequestMapping(value = "startable/set/{processDefinitionId}",
    method = RequestMethod.POST)
@ResponseBody
public String addStartables(@PathVariable("processDefinitionId") String processDefinitionId,
    @RequestParam(value = "users[]", required = false) String[] users,
    @RequestParam(value = "groups[]", required = false) String[] groups) {
    processDefinitionService.setStartables(processDefinitionId, users, groups);
    return "true";
}
```

代码清单14-8 设置流程定义的候选启动人和候选启动组

```
public void setStartables(String processDefinitionId, String[] userArray, String[] groupArray) {
    // 清理现有的设置
    List<IdentityLink> links = repositoryService
        .getIdentityLinksForProcessDefinition(processDefinitionId);
    for (IdentityLink link : links) {
        if (StringUtils.isNotBlank(link.getUserId())) { // 候选人
            repositoryService.deleteCandidateStarterUser(processDefinitionId,
                link.getUserId());
        }
        if (StringUtils.isNotBlank(link.getGroupId())) { // 候选组
            repositoryService.deleteCandidateStarterGroup(processDefinitionId,
                link.getGroupId());
        }
    }
    // 循环添加候选人
    if (!ArrayUtils.isEmpty(userArray)) {
        for (String user : userArray) {
            repositoryService.addCandidateStarterUser(processDefinitionId, user);
        }
    }
    // 循环添加候选组
    if (!ArrayUtils.isEmpty(groupArray)) {
        for (String group : groupArray) {
            repositoryService.addCandidateStarterGroup(processDefinitionId, group);
        }
    }
}
```

14.4.3 读取候选启动数据

单击图14-31中“人[4], 组[1]”后会在记录的下方显示已经设置的候选启动数据，与上一小节设置候选启动数据类似，引擎也提供了对应的读取API，如图14-33所示。

<code>List<IdentityLink></code>	<code>getIdentityLinksForProcessDefinition(String processDefinitionId)</code> Retrieves the <code>IdentityLinks</code> associated with the given process definition.
---------------------------------------	---

图14-33 RepositoryService接口提供的读取候选启动数据API

把流程定义ID作为参数传递给图14-33中的方法即可获取IdentityLink对象集合，然后再根据IdentityLink对象的值判断候选类型是人还是组。在本书配套资源第5章源码包的DeploymentController类的processList()方法中根据查询到的流程定义对象列表循环读取候选启动数据，然后再根据流程定义对象进行分组传递给页面即可。

代码清单14-9 读取流程定义相关的候选启动人和候选组

```
@RequestMapping(value = "/process-list")
public ModelAndView processList(HttpServletRequest request) {
    http://www.hzcourse.com/resource/readBook?path=/openresources/teach_ebook/uncompressed/15030/OEBPS/Text/...
    List<ProcessDefinition> processDefinitionList = processDefinitionQuery
        .listPage(pageParams[0], pageParams[1]);
    http://www.hzcourse.com/resource/readBook?path=/openresources/teach_ebook/uncompressed/15030/OEBPS/Text/...
    // 读取每个流程定义的候选属性
    Map<String, Map<String, List<? extends Object>>> linksMap =
        setCandidateUserAndGroups(processDefinitionList);
    mav.addObject("linksMap", linksMap);
    return mav;
}

private Map<String, Map<String, List<? extends Object>>>
    setCandidateUserAndGroups(List<ProcessDefinition> processDefinitionList) {
    Map<String, Map<String, List<? extends Object>>> linksMap = new HashMap<String, Map<String, List<? extends Object>>>();
    for (ProcessDefinition processDefinition : processDefinitionList) {
        List<IdentityLink> identityLinks = repositoryService
            .getIdentityLinksForProcessDefinition(processDefinition.getId());
        Map<String, List<? extends Object>> single =
            new Hashtable<String, List<? extends Object>>();
        List<User> linkUsers = new ArrayList<User>();
        List<Group> linkGroups = new ArrayList<Group>();
        for (IdentityLink link : identityLinks) {
            if (StringUtil.isNotBlank(link.getUserId())) { // 候选人
                linkUsers.add(identityService.createUserQuery()
                    .userId(link.getUserId()).singleResult());
            } else if (StringUtil.isNotBlank(link.getGroupId())) { // 候选组
                linkGroups.add(identityService.createGroupQuery()
                    .groupId(link.getGroupId()).singleResult());
            }
        }
        single.put("user", linkUsers);
        single.put("group", linkGroups);
        linksMap.put(processDefinition.getId(), single);
    }
    return linksMap;
}
```

最后在展示层从linksMap对象中获取流程定义ID对应的IdentityLink对象集合，循环处理即可显示如图14-31中的数据。下面列出了展示层的代码片段。

```
<tr>
<td>
<ul class="users">
<c:forEach var="user"
items="${linksMap[pd.id]['user']}" >
<li>${user.firstName} ${user.lastName}</li>
</c:forEach>
</ul>
</td>
<td>
<ul class="groups">
<c:forEach var="group"
items="${linksMap[pd.id]['group']}">
<li>${group.name}</li>
</c:forEach>
</ul>
</td>
</tr>
```

14.5 读取引擎属性

引擎提供了一个便捷的接口可以快速获取引擎配置信息，图14-34展示了引擎的相关属性，读取属性的代码如下：

```
Map<String,String> prop = managementService.getProperties();
```



图14-34 引擎配置参数界面

14.6 数据库查询

本书开始时介绍了如何使用H2的Web界面查看引擎的数据库（包括结构及记录），为了方便开发者，引擎提供了查询引擎数据库的接口，用于获取与引擎数据库表的结构、记录。此功能封装在ManagementService接口中，相关API见图14-35所示。

Map<String, Long>	getTableCount() Get the mapping containing {table name, row count} entries of the Activiti database schema.
TableMetaData	getTableMetaData(String tableName) Gets the metadata (column names, column types, etc.) of a certain table.
TablePageQuery	createTablePageQuery() Creates a TablePageQuery that can be used to fetch TablePage containing specific sections of table row data.

图14-35 与数据库查询有关的API

图14-36展示了数据库查询页面的效果图，左侧列出了引擎的所有表名称以及每张表的记录数量，在单击表名后右侧会分页显示表中的记录，具体实现见代码清单14-10所示。

引擎数据库

表名

getTableCount()

ACT_RU_VARIABLE (22)
ACT_RU_EVENT_SUBSCR (0)
ACT_RE_DEPLOYMENT (3)
ACT_RE_PROCDEF (15)
ACT_HI_TASKINST (12)
ACT_HI_ACTINST (58)
ACT_ID_INFO (0)
ACT_ID_MEMBERSHIP (12)
ACT_RU_TASK (6)
ACT_RU_JOB (0)
ACT_GE_BYTEARRAY (40)
ACT_HI_VARINST (34)
ACT_RE_MODEL (0)
ACT_ID_GROUP (7)
ACT_HI_PROCINST (16)
ACT_HI_ATTACHMENT (0)
ACT_ID_USER (12)
ACT_HI_COMMENT (0)
ACT_RU_IDENTITYLINK (21)
ACT_RU_EXECUTION (14)
ACT_HI_DETAIL (18)
ACT_GE_PROPERTY (3)

getTableMetaData("ACT_ID_MEMBERSHIP")

记录 (ACT_ID_MEMBERSHIP)

行	USER_ID_	GROUP_ID_
1	henry	admin
2	bill	generalManager
3	jenny	hr
4	kermit	deptLeader
5	andy	deptLeader
6	tony	treasurer
7	tony	cashier
8	thomas	supportCrew
9	amy	cashier
10	amy	deptLeader

共12条数据 << < 1 2 > >>

createTablePageQuery().tableName(tableName).listPage(1, 10)

图14-36 引擎的数据库（表结构及记录）

代码清单14-10 读取引擎数据库

```
@Controller
@RequestMapping("/chapter14/database")
public class DatabaseController {
    @Autowired
    ManagementService managementService;
    @RequestMapping("")
    public ModelAndView index(@RequestParam(value = "tableName", required = false)
        String tableName, HttpServletRequest request) {
        ModelAndView mav = new ModelAndView("chapter14/database");
        // 读取表（表名，记录数量）
        Map<String, Long> tableCount = managementService.getTableCount();
        mav.addObject("tableCount", tableCount);
        // 读取表记录，在页面单击左边的表名后传递tableName参数
        if (StringUtils.isNotBlank(tableName)) {
            TableMetaData tableMetaData = managementService
                .getTableMetaData(tableName);
            mav.addObject("tableMetaData", tableMetaData);
            Page<Map<String, Object>> page = new Page<Map<String, Object>>(10);
            int[] pageParams = PageUtil.init(page, request);
            TablePage tablePages = managementService.createTablePageQuery()
                .tableName(tableName).listPage(pageParams[0], pageParams[1]);
            page.setResult(tablePages.getRows());
            page.setTotalCount(tableCount.get(tableName));
            mav.addObject("page", page);
        }
        return mav;
    }
}
```

TableMetaData对象可以获取指定表中的字段名称（getColumnName()方法）及字段类型（getColumnTypes()方法）集合对象。从TablePage对象中可以通过调用getRows()方法获取记录集合，类型为Map<String, Object>，这样可以根据字段名称获取对应的值。读者可以参考database.jsp查看展示层的实现。

14.7 用户与组

对于如何通过API管理用户、组在第5章已经介绍过了，本章的主题是管理员特性，所以关于用户、组的管理是必不可少的内容。本节不再介绍相关API的使用。本章的随书配套资源中提供了完整的演示，读者可以参考本章源码的IdentityController.java与user-list.jsp及group-list.jsp。用户和组的管理效果如图14-37和图14-38所示。

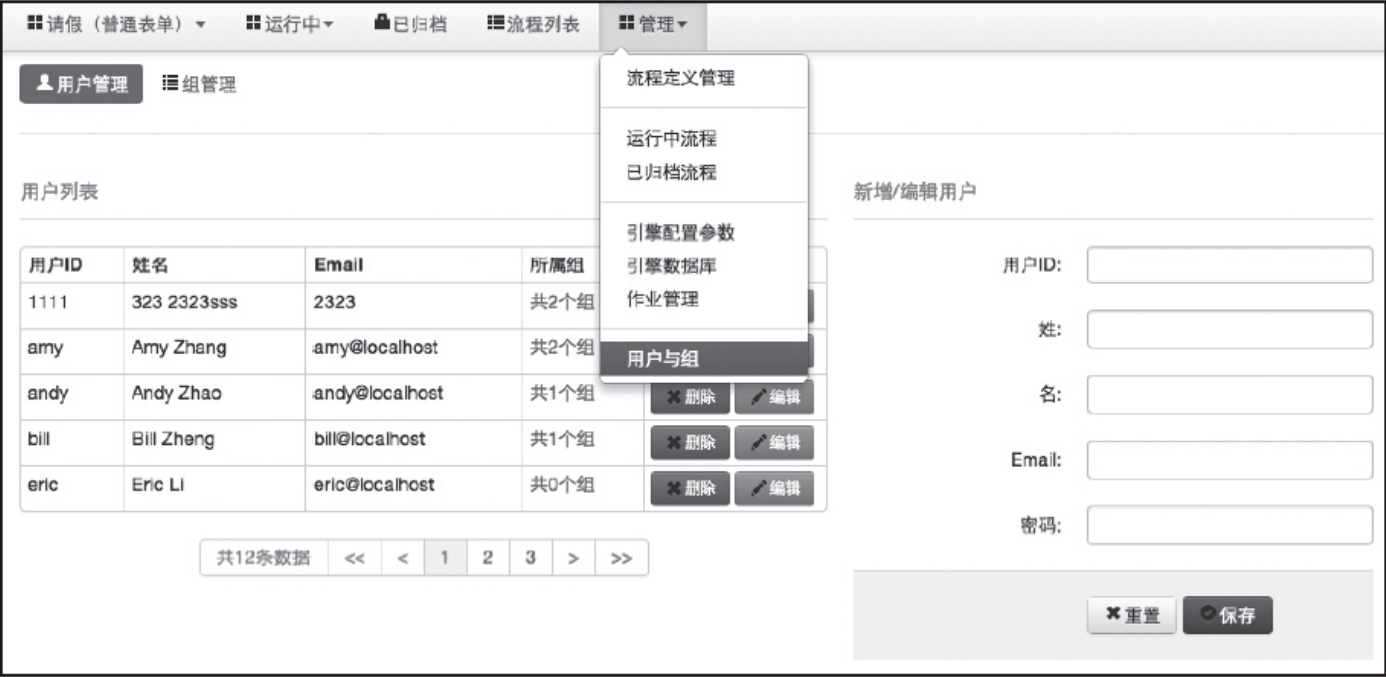


图14-37 用户管理界面



图14-38 组管理界面

14.8 本章小结

本章作为实战篇的最后一章对所有与管理员有关的功能特性进行了详细的讲解，其中涉及的管理员特性可以让企业的系统管理员全权管理流程数据。

管理员除了能够查询引擎中的所有数据之外，还可以控制流程及流程定义的状态（挂起、激活），当然读者在实际操作中也可以把状态控制操作移植到任务办理界面（切记控制好操作权限）。另外控制流程定义权限可以避免越权操作，减少垃圾数据的产生。

作业管理是引擎中一个比较重要的功能，除了管理各种定时事件产生的作业之外，还要承担异步类型的作业管理任务，因此管理员可以通过该模块干预作业的执行、查看作业执行结果，遇到作业执行失败时可以把作业的异常信息报告给IT部门或者供应商。在14.2.3节中对流程任务的独占与异步执行进行了详细的分析，并且分析了乐观锁问题的原因并给出了如何绕过乐观锁问题的解决办法。

最后简单介绍了如何读取引擎的数据库信息（结构、记录）及引擎属性。

第四部分 高级篇

相信经过前面的学习，大家已经掌握了Activiti的大部分功能以及应用场景，可以根据需求设计流程图，根据业务逻辑控制流程的走向，基本上可以满足企业应用中80%的基本需求了。

接下来的部分为高级篇，是Activiti作为一个BPM Platform（BPM平台）不可或缺的一部分内容。与企业现有系统或功能模块集成，把Activiti作为一个业务流程的调度中心，让多个系统各司其职，用流程把复杂的业务串联起来，当某个环节需要内部或外部系统提供服务时可以利用引擎的内置或扩展的组件与具体模块交互，从而完成流程中的交互任务。

这部分的内容包括集成Webservice、规则引擎、ESB（Enterprise Service Bus，企业服务总线）、JPA（可以把流程变量保存在数据库的表中），以及可以用CDI方式注册引擎组件。

第15章 集成WebService

在企业应用中，与外界系统交互可以选择WebService、Socket、HTTP协议或者类WebService（早期没有WebService时通信双方先组装再解析XML）方式。目前大多数系统都是通过SOAP（Simple Object Access Protocol，简单对象访问协议）方式，它提供了标准的RPC（Remote Procedure Call Protocol，远程过程调用协议）方法调用WebService。

在第4章中提到，WebService任务不是BPMN 2.0规范的一部分，而是由Activiti在ServiceTask的基础上扩展而来的（读者也可以自己扩展实现，后面的章节会详细介绍）。只要在流程中引入WebService接口的XML定义以及相关的参数就可以通过WebService方式交互，当然交互的结果完全可以作为流程变量存在并可以在网关中作为表达式的一部分控制流程的走向。

本章以请假流程（动态表单）为例演示如何在流程中集成WebService任务控制流程走向，对应的流程图如图15-1所示。

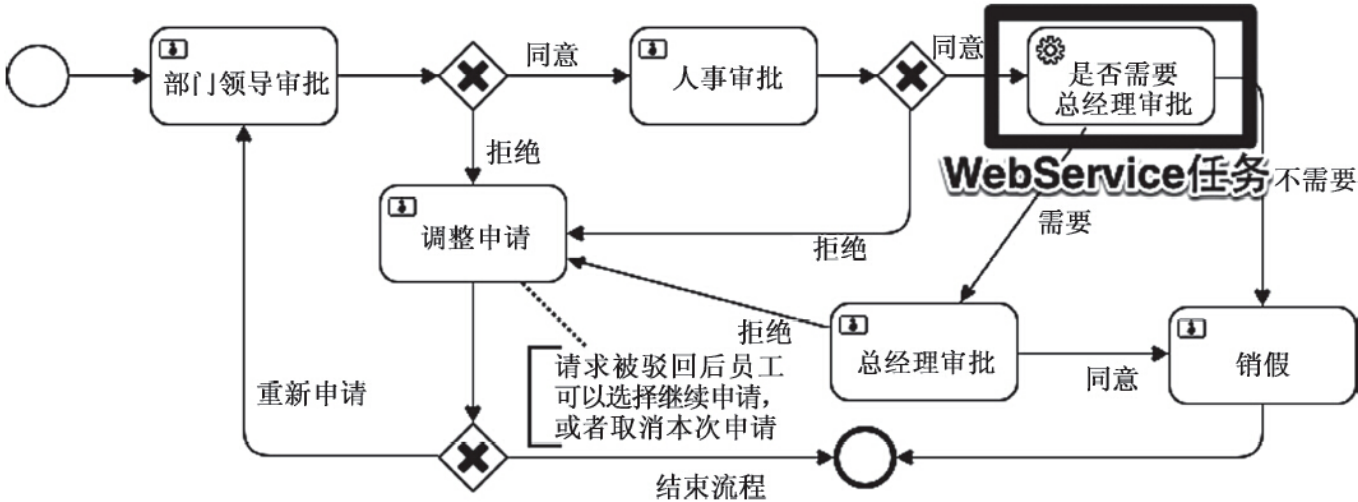


图15-1 包含了WebService任务的请假流程

与第6章的请假流程相比，图15-1中新增了两个任务：是否需要总经理审批、总经理审批，前者提供业务逻辑判断，当申请的天数大于3天时任务流转至“总经理审批”，否则流转至“销假”。

15.1 发布WebService服务

要提供WebService服务，首先要创建并发布一个可以提供请假天数判断的WebService，这里我们使用Apache CXF（http://cxf.apache.org）框架发布WebService服务。代码清单15-1列出了一个标准的LeaveWebService接口定义。

代码清单15-1 LeaveWebService接口

```
package me.kafeitu.activiti.chapter15.leave.ws;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;
@WebService
public interface LeaveWebService {
    /**
     * 是否需要总经理审批
     * @param startDate 请假开始时间
     * @param endDate 请假结束时间
     * @return true/false
     */
    @WebResult(name="needed")
    boolean generalManagerAudit(@WebParam(name = "startDate") String startDate,
                                @WebParam(name = "endDate") String endDate);
}
```

接着定义一个LeaveWebService接口的实现类LeaveWebServiceImpl，见代码清单15-2所示。

代码清单15-2 LeaveWebService接口实现类LeaveWebServiceImpl

```
package me.kafeitu.activiti.chapter15.leave.ws;
@WebService(endpointInterface = "me.kafeitu.activiti.chapter15.leave.ws.LeaveWebService", serviceName = "LeaveWebService")
public class LeaveWebServiceImpl implements LeaveWebService {
    // 计算开始日期与结束日期相差天数
    public int daysBetween(Date startDate, Date endDate) {
        JDatetime joddStartDate = new JDatetime(startDate);
        JDatetime joddEndDate = new JDatetime(endDate);
        int result = joddStartDate.daysBetween(joddEndDate);
        return result;
    }
    @Override
    public boolean generalManagerAudit(String startDate, String endDate) {
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm");
        int days = 0;
        try {
            days = daysBetween(sdf.parse(startDate), sdf.parse(endDate));
        } catch (ParseException e) {
            logger.error("解析日期出错: startDate={}, endDate={}", startDate, endDate);
        }
        return days > 3 ? true : false;
    }
}
```

代码清单15-2简单实现了LeaveWebService接口，判断请假天数超过三天时返回true，否则返回false。

为了使接口的实现类LeaveWebServiceImpl的逻辑符合我们的预期，有必要为这个接口实现进行单元测试。假设发布的WebService接口的URL定义为“http://localhost:9999/leave”，当然真实应用应该是一个IP地址或者域名，本例为了演示暂且使用localhost作为服务的地址。

为了方便测试，封装了一个LeaveWebService接口的工具类LeaveWebServiceUtil，可以方便启动、停止WebService服务，见代码清单15-3所示。

代码清单15-3 LeaveWebService接口工具类LeaveWebServiceUtil

```
public class LeaveWebserviceUtil {
    public static final String WEBSERVICE_URL = "http://localhost:9999/leave";
    public static final String WEBSERVICE_WSDL_URL = WEBSERVICE_URL + "?wsdl";
    public static final String WEBSERVICE_URI =
        "http://ws.leave.chapter15.activiti.kafeitu.me/";
    private static Server server = null;
    // 发布WebService
    public static void startServer() {
        LeaveWebService leaveWebService = new LeaveWebServiceImpl();
        // 也可以用Apache CXF方式发布
        Endpoint.publish(WEBSERVICE_URL, leaveWebService);
        System.out.println("请假Webservice已发布: " + WEBSERVICE_URL + "?wsdl");
    }
    // 停止Webservice服务
    public static void stopServer() {
        if (server != null) {
            server.destroy();
        }
    }
    public static void main(String[] args) {
        startServer();
    }
}
```

在调用LeaveWebServiceUtil的main方法或者startServer方法之后，控制台打印了如图15-2所示的输出，表示WebService服务已经成功发布，现在在浏览器中访问“<http://localhost:9999/leave?wsdl>”后就可以看到如图15-3所示的XML定义。

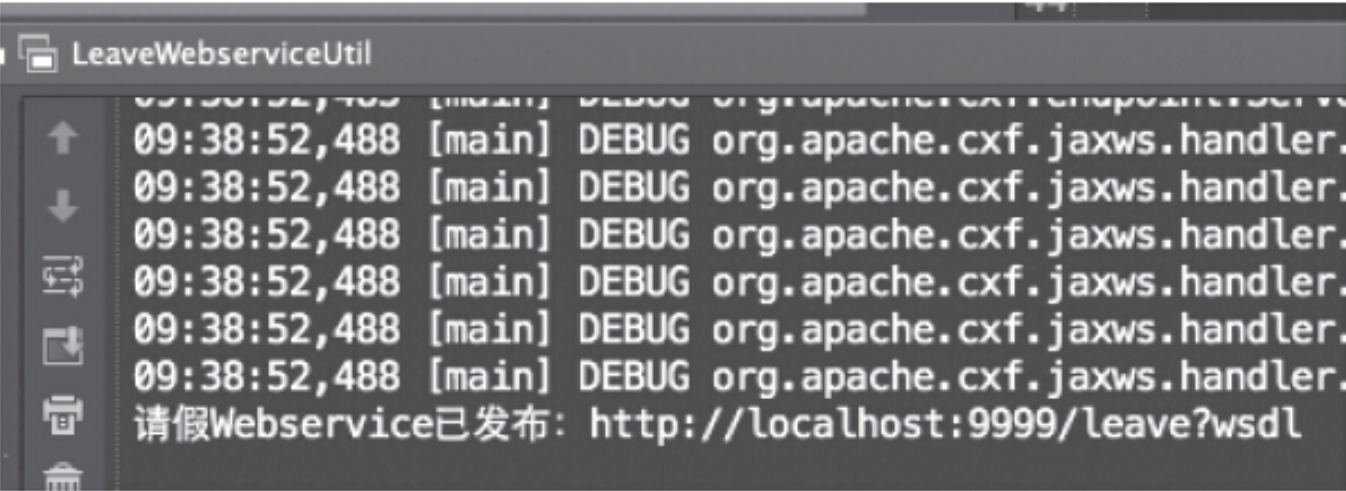


图15-2 通过LeaveWebServiceUtil工具类发布了WebService服务



图15-3 LeaveWebService接口的WSDL内容

接着编写一个简单的单元测试来验证判断请假天数的逻辑，测试类名称为LeaveWeb-ServiceBusinessTest，见代码清单15-4所示。

代码清单15-4LeaveWebServiceBusinessTest单元测试

```

public class LeaveWebServiceBusinessTest {
    // 需要总经理审批
    @Test
    public void testTrue() throws ParseException {
        JaxWsProxyFactoryBean factory = new JaxWsProxyFactoryBean();
        factory.getInInterceptors().add(new LoggingInInterceptor());
        factory.getOutInterceptors().add(new LoggingOutInterceptor());
        factory.setServiceClass(LeaveWebService.class);
        factory.setAddress(LeaveWebServiceUtil.WEBSERVICE_URL);
        LeaveWebService leaveWebService = (LeaveWebService) factory.create();
        boolean audit = leaveWebService.generalManagerAudit(
            "2013-01-01 09:00", "2013-01-05 17:30");
        assertTrue(audit);
    }
}

```

从代码清单15-4中可以看出，从2013-01-01请假到2013-01-05肯定大于3天，所以WebService返回的结果应该是“true”。当然这里的判断不严谨，在实际应用中还要判断上下班的边界值，读者可以自行扩展。很显然单元测试可以顺利通过，如图15-4所示。

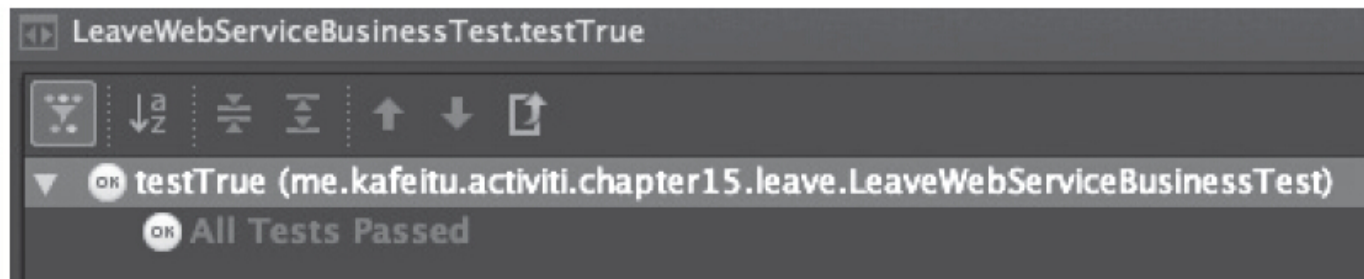


图15-4 LeaveWebServiceBusinessTest单元测试的执行结果

抱歉，这里“浪费”了几分钟时间做了一些准备工作，创建并发布了可用WebService，并且对健壮性进行了简单的测试，这样就没有了后顾之忧，下面可以切入正题了。

15.2 在流程中定义WebService任务

图15-1中新添加的“是否需要总经理审批”Service任务要实现WebService服务的调用需要以下几个步骤：

- 引入WebService的Scheme。
- 声明interface。
- 定义message。
- 定义输入、输出Data。

在阅读本节内容时可以结合本书配套资源中本章示例代码（bpmn20-example项目）的/chapter15/leave-webservice.bpmn文件。

从文件头部的definitions标签开始，加入如下属性定义：

```
targetNamespace="me.kafeitu.activiti.chapter15.leave.ws"
xmlns:tns="me.kafeitu.activiti.chapter15.leave.ws"
xmlns:leave="http://webservice.kafeitu.me"
```

紧接着引入WSDL，在流程运行时引擎从WSDL中获取WebService的定义，然后把该地址交给Apache CXF处理。

```
<import importType=http://schemas.xmlsoap.org/wsdl/
location="http://localhost:9999/leave?wsdl"
namespace="http://webservice.kafeitu.me/" />
```

importType属性的值是WSDL的Scheme引用（必须定义），location是实际应用中已经发布的WebService的WSDL地址，namespace属性的值与“xmlns:leave”保持一致。

下面的代码列出了interface的定义，用来声明WebService接口的operation以便“ServiceTask”中可以引用定义的接口（operation的id属性）。

```
<!-- interface: implementationRef属性的值要与WSDL (wsdl:portType) 中的保持一致，即@WebService中的serviceName属性 -->
<interface name="Leave Audit Interface"
implementationRef="leave:LeaveWebservice">
  <!-- operation: implementationRef属性的值也要与WSDL中的保持一致 (wsdl:operation) -->
  <operation id="auditOperation"
name="General Manager Audit Operation"
implementationRef="leave:generalManagerAudit">
    <inMessageRef>
      tns:generalManagerAuditRequestMessage
    </inMessageRef>
    <outMessageRef>
      tns:generalManagerAuditResponseMessage
    </outMessageRef>
  </operation>
</interface>
<!-- 定义message，参考wsdl文件中的wsdl:message的定义 -->
<message id="generalManagerAuditRequestMessage"
itemRef="tns:generalManagerAuditRequestItem" />
<message id="generalManagerAuditResponseMessage"
itemRef="tns:generalManagerAuditResponseItem" />
<!-- 参考wsdl:input -->
<itemDefinition id="generalManagerAuditRequestItem"
structureRef="leave:generalManagerAudit" />
<!-- 参考wsdl:output -->
<itemDefinition id="generalManagerAuditResponseItem"
structureRef="leave:generalManagerAuditResponse" />
```

对于上面的代码，可以结合图15-3中的XML理解，虽然标签比较多，但是都有层级、引用关系，根据xxxRef属性跟踪即可。

基于ServiceTask定义WebService任务需要设置“implementation”这个特殊的属性，且其固定值定义为“##WebService”，再通过“operationRef”属性定义触发哪个WebService操作。

```
<!-- 定义变量 -->
<itemDefinition id="needed" structureRef="boolean"/>
<itemDefinition id="startDate" structureRef="string"/>
<itemDefinition id="endDate" structureRef="string"/>
<serviceTask id="checkGeneralManagerAudit"
name="是否需要总经理审批"
implementation="##WebService"
operationRef="tns:auditOperation">
  <ioSpecification>
    <dataInput id="dataInput"
itemSubjectRef="tns:generalManagerAuditRequestItem"/>
    <dataOutput id="dataOutput"
itemSubjectRef="tns:generalManagerAuditResponseItem"/>
    <inputSet>
      <dataInputRefs>dataInput</dataInputRefs>
    </inputSet>
    <outputSet>
      <dataOutputRefs>dataOutput</dataOutputRefs>
    </outputSet>
  </ioSpecification>
  <dataInputAssociation> // 输入数据
    <sourceRef>startDate</sourceRef>
    <targetRef>startDate</targetRef>
  </dataInputAssociation>
  <dataInputAssociation>
    <sourceRef>endDate</sourceRef>
    <targetRef>endDate</targetRef>
  </dataInputAssociation>
  <dataOutputAssociation> // 输出数据
    <sourceRef>needed</sourceRef>
    <targetRef>needed</targetRef>
  </dataOutputAssociation>
</serviceTask>
```

用ioSpecification标签定义输入输出对象。用dataInputAssociation定义接口的输入参数（来源与流程变量）。用dataOutputAssociation定义接口的输出结果（保存到流程变量中）：其中sourceRef和targetRef必须设置，即使这两个值相同也不可省略，定义的名称需要用“itemDefinition”预先定义，用属性“structureRef”定义变量的类型，该属性允许为空不过为了显示地标记变量的类型可以用Java的类型进行定义，例如示例中的“string”。

15.3 在流程中调用WebService

大概了解了XML如何定义及多个标签之间的关系，再结合代码清单15-5的单元测试执行结果进行分析，相信读者很快就明白这其中的奥秘。

不过在运行单元测试之前需要先把activiti-cxf模块加入到项目中，在项目的pom.xml中添加如下配置，activiti-cxf模块会依赖Apache CXF，如果实际项目已经依赖了Apache CXF，要注意版本号的差异。

```
<dependency>
    <groupId>org.activiti</groupId>
    <artifactId>activiti-cxf</artifactId>
    <version>${activiti.version}</version>
</dependency>
```

代码清单15-5 LeaveWebserviceInWorkflowTest单元测试

```
public class LeaveWebserviceInWorkflowTest extends AbstractTest {
    // 调用Webservice后决定是否需要总经理审批
    @Test
    @Deployment(resources = "chapter15/leave-webservice.bpmn")
    public void testWithWebservice() throws Exception {
        ProcessDefinition processDefinition =
            repositoryService.createProcessDefinitionQuery()
                .processDefinitionKey("leave-webservice").singleResult();
        // 启动流程（请假时间段间隔4天）
        http://www.hzcourse.com/resource/readBook?path=/openresources/teach_ebook/uncompressed/15030/OEBPS/Text/...
        // 部门领导审批通过
        http://www.hzcourse.com/resource/readBook?path=/openresources/teach_ebook/uncompressed/15030/OEBPS/Text/...
        // 人事审批通过
        http://www.hzcourse.com/resource/readBook?path=/openresources/teach_ebook/uncompressed/15030/OEBPS/Text/...
        // 判断Webservice任务执行结果
        Boolean needed = (Boolean) runtimeService
            .getVariable(processInstance.getId(), "needed");          #1
        assertTrue(needed);
    }
}
```

由于代码清单15-5的单元测试需要动态读取WebService的WSDL，因此再运行单元测试只要先执行LeaveWebserviceUtil.startServer()方法发布WebService。

从图15-5中可以看出通过了单元测试，LeaveWebService中的generalManagerAudit方法返回了名称为“needed”结果，在流程文件中又把该结果设置到同名的变量中，所以代码清单15-5中的#1处查询名称为“needed”的变量结果为true。

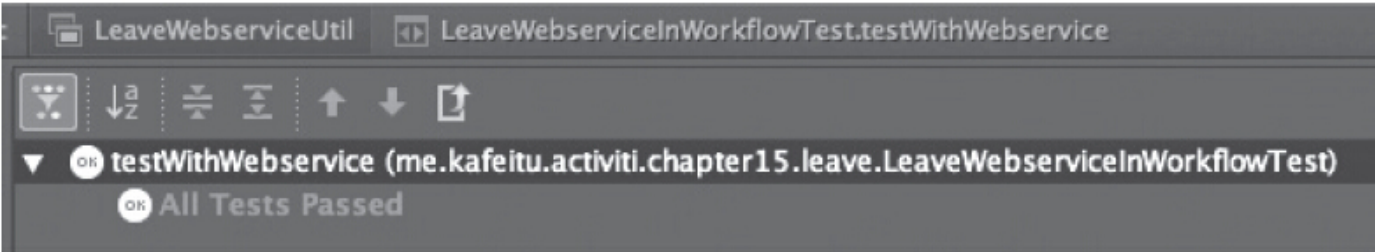


图15-5 代码清单15-5的单元测试执行成功

15.4 本章小结

本章介绍了企业应用中与外界系统交互的一种方式——WebService。WebService任务并未包含在BPMN 2.0规范中，而是Activiti基于serviceTask扩展的一个自定义任务类型。引擎对WebService任务提供了基础的功能支持，可以把调用外部WebService任务作为流程的一个步骤，从而更有利于企业流程的规划。

本章首先介绍了如何发布一个简单的WebService服务，然后又介绍了如何把发布的WebService引入到流程定义文件中，最后介绍了如何把流程数据传递给WebService，以及如何获取WebService接口返回的处理结果。

第16章 集成规则引擎

规则引擎可以像流程引擎一样嵌入在应用中。利用规则引擎可以将业务规则脱离应用单独维护，使用预定义的规则文件语法编写业务规则，把业务数据交给规则引擎解释，然后返回处理结果，根据返回的结果可以判定业务的处理方式，有点类似BPMN 2.0规范中的路由功能。

Activiti目前对BPMN 2.0规范中的“BusinessRuleTask”做了基本的实现，使用了JBoss公司的规则引擎——Drools。Drools也是目前比较流行的规则引擎之一，JBoos公司发布的jBPM5引擎也是在Drools的基础上开发的。

在15.1节的请假流程中，利用WebService判定请假天数是否超过3天以决定是否需要总经理审批，本节将利用规则引擎完成相同的功能。

16.1 定义流程与规则

图16-1与图15-1类似，只不过在图16-1中把WebService任务换成了BusinessRule-Task（是否需要总经理审批），可以在流程设计器中配置一些规则有关的参数，如图16-2所示。

图16-2的配置对应的XML如下所示，对XML的定义可以参考4.3.5节对业务规则任务的属性说明。

```
<businessRuleTask id="checkGeneralManagerAudit"
    name="是否需要总经理审批"
    activiti:rules="检查是否需要总经理审批"
    activiti:ruleVariablesInput="{leave}"
    activiti:resultVariable="rulesOutput" />
```

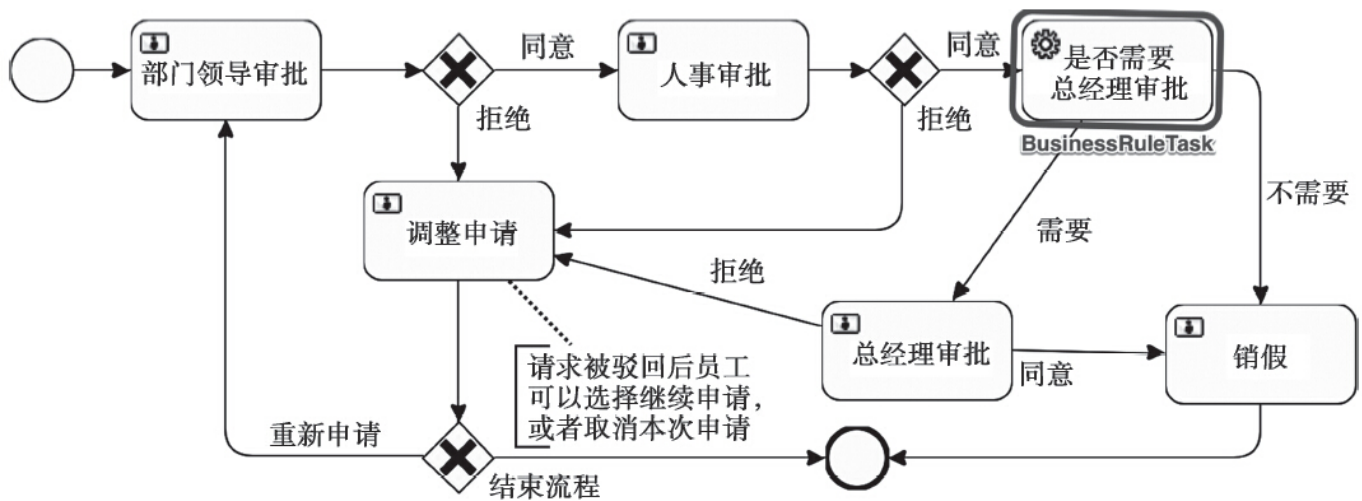


图16-1 包含BusinessRuleTask的请假流程

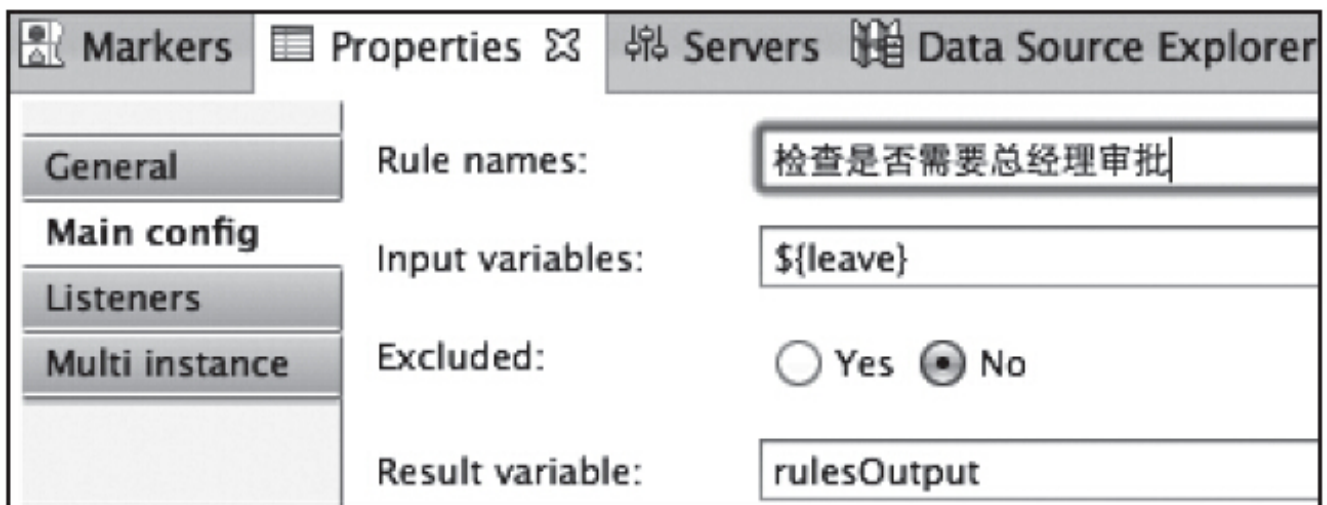


图16-2 图16-1中“是否需要总经理审批”任务的属性配置

把XML转换成文字可以这么理解，把参数“\${leave}”传递给名称为“检查是否需要总经理审批”的规则（rule），在执行完规则后把结果保存到变量“rulesOutput”中。

代码清单16-1列出了判定请假天数的规则定义，具体可以查看bpmn20-example/test/resources/leave.drl文件。

代码清单16-1 请假规则文件定义

```
package me.kafeitu.activiti.chapter15;
import me.kafeitu.activiti.chapter15.drools.Leave;
import jodd.datetime.JDateTime;
import org.activiti.engine.impl.persistence.entity.ExecutionEntity;
rule "检查是否需要总经理审批"
when
    $leave: Leave(); // 接收从引擎传递的变量，根据类型匹配
then
    JDateTime joddStartDate = new JDateTime($leave.getStartTime());
    JDateTime joddEndDate = new JDateTime($leave.getEndTime());
    int result = joddStartDate.daysBetween(joddEndDate);
    // 请假天数大于3天
    $leave.setGeneralManagerAudit(result > 3);
end
```

16.1.1 部署规则文件

要调用规则，首先要把规则文件部署到引擎数据库中，规则文件扩展名为“.drl”。Activiti默认不支持该类型的文件，需要在引擎配置对象中配置一个可以解析规则文件的部署器，配置如下：

```
<bean id="processEngineConfiguration"
class="org.activiti.engine.impl.cfg
.StandaloneInMemProcessEngineConfiguration">
<!-- 部署drl文件时需要设置一个自定义的部署解析类 -->
<property name="customPostDeployers">
<list>
<bean class="org.activiti.engine
.impl.rules.RulesDeployer" />
</list>
</property>
</bean>
```

在上面的配置中设置了引擎配置对象的“customPostDeployers”属性，添加了支持规则文件的部署器，把规则文件交给Drools解析处理，这样在流程运行时引擎可以调用Drools的API执行规则。该部署器实现了org.activiti.engine.impl.persistence.deploy.Deployer接口，如果需要自定义部署器，可以实现该接口并添加实现类到“customPost-Deployers”属性中。

16.1.2 单元测试

代码清单16-2列出了包含业务规则任务的单元测试代码。

代码清单16-2 LeaveWithBusinessRuleTest单元测试

```
public class LeaveWithBusinessRuleTest extends AbstractTest {
    @Test
    @Deployment(resources = {"chapter15/leave-drools.bpmn", "chapter15/leave.drl"})
    public void testWithWebservice() throws Exception {
        ProcessDefinition processDefinition =
            repositoryService.createProcessDefinitionQuery()
                .processDefinitionKey("leave- drools").singleResult();
        Leave leave = new Leave();
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm");
        Map<String, String> variables = new HashMap<String, String>();
        Calendar ca = Calendar.getInstance();
        leave.setStartTime(ca.getTime());
        String startDate = sdf.format(ca.getTime());
        ca.add(Calendar.DAY_OF_MONTH, 4); // 当前日期加4天
        leave.setEndTime(ca.getTime());
        // 启动流程（请假时间段间隔4天）
        runtimeService.setVariable(processInstance.getId(), "leave", leave);
        // 部门领导审批通过
        http://www.hzcourse.com/resource/readBook?path=/openresources/teach_ebook/uncompressed/15030/OEBPS/Text/...
        // 人事审批通过
        http://www.hzcourse.com/resource/readBook?path=/openresources/teach_ebook/uncompressed/15030/OEBPS/Text/...
        // 判断drools执行结果
        Leave leaveOutput = (Leave) runtimeService
            .getVariable(processInstance.getId(), "leave");
        assertTrue(leaveOutput.isGeneralManagerAudit());
        // 返回的结果集
        // 如果没有设置activiti:resultVariable属性默认的名称为：
        // org.activiti.engine.rules.OUTPUT
        Collection<Object> ruleOutputList = (Collection<Object>)
            runtimeService.getVariable(processInstance.getId(), "rulesOutput");
        assertNotNull(ruleOutputList);
        assertEquals(1, ruleOutputList.size());
        leave = (Leave) ruleOutputList.iterator().next();
        assertTrue(leave.isGeneralManagerAudit());
    }
}
```

代码清单16-2对流程做了一个简单的测试，期间会调用规则文件（见代码清单16-1），最后验证执行结果（流程流转至总经理审批）。

在#1处除了部署流程文件（leave-drools.bpmn）之外，还部署了规则文件（leave.drl），当解析leave.drl文件时将规则文件内容交给org.activiti.engine.impl.rules.RulesDeployer类处理。

在#2处创建了一个Leave对象用于设置请假的时间及保存规则处理结果。

在#3处设置了流程变量“leave”，也就是流程定义文件中属性“activiti:ruleVariables-Input”的表达式中需要使用的变量。

在#4处从流程中读取变量检查规则执行结果，获取“leave”变量后检查general-ManagerAudit属性为true（#4-E处）。需要注意的是#4-S处获取的leave对象与#3处设置的leave对象不是同一个，在规则文件执行完成后引擎会把Drools返回的对象。

16.2 本章小结

本章实现的功能与上一章相同——判断请假的时间是否超过3天，但是方法不同。WebService任务依赖外部的服务来完成判断，本章则利用BPMN 2.0规范的Business Rule任务完成。利用JBoos公司的Drools作为规则引擎，把流程文件与规则定义文件一同部署到Activiti中即可在流程中调用。

需要注意的是，引擎默认不支持规则定义文件的部署、解析，所以当使用规则定义文件时要在引擎配置中注册规则来部署实现类。

第17章 集成JPA

JPA（Java Persistence API）在大多数系统中已经得到广泛应用，从1.0标准到目前的2.0标准，越来越多的开源框架发布了自己的JPA实现，例如Hibernate、Open JPA、Spring Data等。

在Activiti中使用MyBatis作为ORM层，读者可能会问JPA和Activiti有什么关系？

让我们先回顾一下动态表单、外置表单的一些特点，所有的表单数据都保存在引擎的一张表中，然而时间长了之后数据会越来越大，而且数据没有结构很难查询、优化，这也是有些企业不选择使用这两种表单的原因，在Activiti中集成JPA就是为了解决这个问题。

在Activiti集成JPA后可以把动态表单或外置表单产生的“行”式存储转换为“列”式存储，也就是把业务数据保存在用户自定义的表中，这样有利于数据的查询。

17.1 配置JPA

我们知道，引擎使用MyBatis作为ORM。可能的读者会有疑问：“在集成JPA后，JPA会与MyBatis有冲突吗？”答案肯定是没有，因为两者是完全分离的，使用JPA都是面向接口编程的，在引擎内部只是做了一些封装、集成的工作，并且对MyBatis和JPA的事务统一管理。

表17-1列举了引擎配置对象中有关JPA的属性及其含义。

表17-1 Activiti引擎配置对象中有关JPA的属性及其含义

属 性 名 称	属 性 说 明
jpaPersistenceUnitName	持久化单元的名称（要确保该持久化单元在类路径下是可用的）。根据该规范，默认的路径是 /META-INF/persistence.xml）。该属性与 jpaEntityManagerFactory 选择一个即可
jpaEntityManagerFactory	一个实现了 javax.persistence.EntityManagerFactory 的 bean 的引用（例如 spring-data-jpa 模块提供的 org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean）。该属性与 jpaPersistenceUnitNam 选择一个即可
jpaHandleTransaction	是否由引擎来管理 JPA 的事务，一般情况下使用 Spring、JTA 或由容器提供的统一事务管理功能把此属性设置为 false，Standalone 模式设置为 true
jpaCloseEntityManager	表示流程引擎是否应该关闭从 EntityManagerFactory 获取的 EntityManager 的实例，和 jpaHandleTransaction 类似，如果使用了容器管理事务设置为 false，否则设置为 true

17.1.1 Standalone模式

在引擎中集成JPA很简单，只需要在引擎配置对象中设置持久化单元（Persistence Unit）名称即可。代码清单17-1列出了Standalone模式配置JPA的代码片段。

代码清单17-1 在引擎Standalone模式中配置JPA

```
<bean id="processEngineConfiguration" class="org.activiti.engine.impl.cfg.StandaloneInMemProcessEngineConfiguration">
http://www.hzcourse.com/resource/readBook?path=/openresources/teach_ebook/uncompressed/15030/OEBPS/Text/...
  <property name="jpaPersistenceUnitName" value="aia-jpa-pu" /> #1
  <property name="jpaHandleTransaction" value="true" />
  <property name="jpaCloseEntityManager" value="true" />
http://www.hzcourse.com/resource/readBook?path=/openresources/teach_ebook/uncompressed/15030/OEBPS/Text/...
</bean>
```

17.1.2 Spring模式

相信目前大多数应用都是使用Spring来统一管理事务，所以这里着重介绍一下使用Spring代理引擎配置对象时如何集成JPA，稍后的示例也是基于Spring模式进行讲解的。

代码清单17-2列出了在Spring模式中配置JPA。

代码清单17-2 在引擎Spring模式中配置JPA

```
<bean id="hibernateJpaVendorAdapter" class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
  <property name="databasePlatform">
    <bean factory-method="getDialect" class="me.kafeitu.activiti.utils.Hibernates">
      <constructor-arg ref="dataSource"/>
    </bean>
  </property>
</bean>
<bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <property name="dataSource" ref="dataSource"/>
  <property name="jpaVendorAdapter" ref="hibernateJpaVendorAdapter"/>
  <property name="packagesToScan" value="me.kafeitu.activiti"/>
  <property name="jpaProperties">
    <prop key="hibernate.show_sql">true</prop> // 打开显示SQL开关
    <prop key="hibernate.format_sql">true</prop> //由Hibernate格式化引擎执行的SQL
  </property>
</bean>
<bean id="hibernateJpaVendorAdapter" class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
</bean>
<!-- Spring Data Jpa配置 -->
<jpa:repositories base-package="me.kafeitu.activiti" transaction-manager-ref="transactionManager"
  entity-manager-factory-ref="entityManagerFactory"/>
<!-- JPA 事务配置 -->
<bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">
  <property name="entityManagerFactory" ref="entityManagerFactory"/>
</bean>
<bean id="processEngineConfiguration" class="org.activiti.spring.SpringProcessEngineConfiguration">
  <property name="dataSource" ref="dataSource" />
  <property name="transactionManager" ref="transactionManager" />
  ...
  <!-- JPA -->
  <property name="jpaEntityManagerFactory" ref="entityManagerFactory" /> #1
  <property name="jpaHandleTransaction" value="false" /> #2
  <property name="jpaCloseEntityManager" value="false" /> #3
</bean>
```

对比代码清单17-1-与代码清单17-2，代码清单17-2中除了必要的EntityManager-Facory及事务管理器之外，最大的区别在于#2、#3处的两个属性的值均设置为false，意思就是由容器（这里指的是Spring）管理事务且不关闭EntityManager实例。

17.2 JPA版本的请假流程

在之前的章节中我们接触到了不同版本的请假流程，本章内容将以动态表单版本的请假流程为基础进行改造，目的是在流程启动时可以自动创建实体对象并保存到数据库表，流程图如图17-1所示。

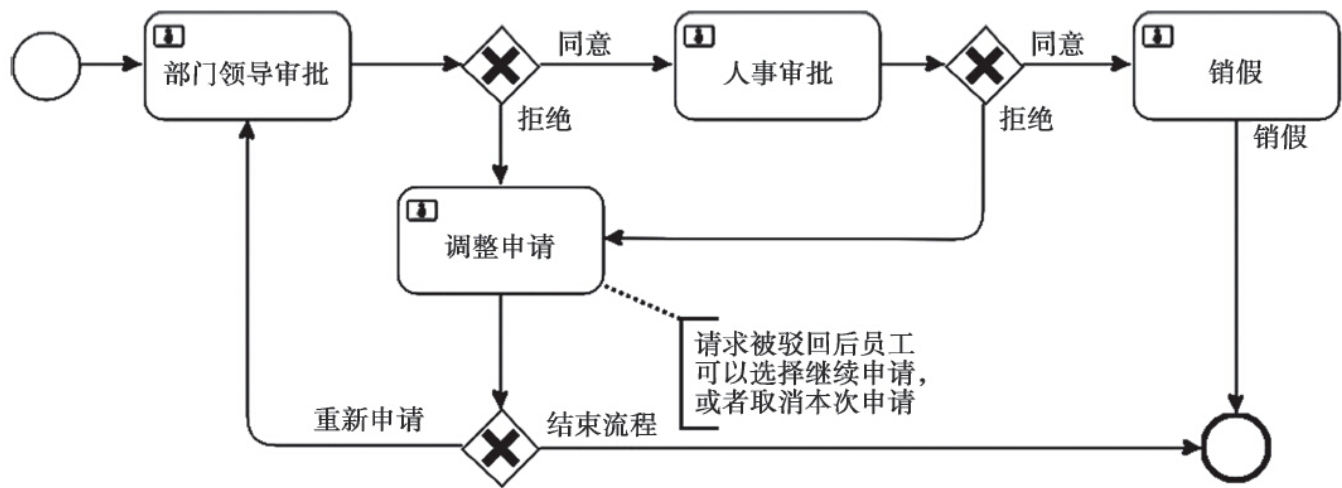


图17-1 请假流程图

改造的内容如下：

- 1) 流程启动时创建请假实体并持久化到数据库。
- 2) 保存部门领导审批意见到请假表 (AIA_C17_LEAVE) 。
- 3) 保存人事审批意见到请假表。
- 4) 保存销假节点填写的内容。

在第7章介绍普通表单时创建了表“LEAVE”，为了和之前有所区分，所以本章会使用“AIA_C17_LEAVE”作为新的数据表，在“LEAVE”的基础上添加了两个字段（部门领导审批结果、人事审批结果），如图17-2所示。

show columns from aia_c17_leave;

FIELD	TYPE
ID	BIGINT(19)
APPLY_TIME	TIMESTAMP(23)
DEPT_LEADER_APPROVED	VARCHAR(255)
END_TIME	TIMESTAMP(23)
HR_APPROVED	VARCHAR(255)
LEAVE_TYPE	VARCHAR(255)
PROCESS_INSTANCE_ID	VARCHAR(255)
REALITY_END_TIME	TIMESTAMP(23)
REALITY_START_TIME	TIMESTAMP(23)
REASON	VARCHAR(255)
REPORT_BACK_DATE	TIMESTAMP(23)
START_TIME	TIMESTAMP(23)
USER_ID	VARCHAR(255)

图17-2 AIA_C17_LEAVE表结构

 **提示** 表AIA_C17_LEAVE会在启动本章示例时由Hibernate自动创建。

单从流程图上看不出与前面章节接触到的请假流程有任何区别，本章内容只是对现有流程进行了局部改造，添加了一些监听器而已，稍后详解介绍。流程文件位于本章示例（chapter17-jpa）的“src/main/resources/chapter17”目录下，读者可以结合代码学习。

17.2.1 启动流程时持久化JPA实体

对于请假流程，在启动流程时需要申请人填写表单，在启动流程后我们需要把这些表单内容持久化到自定义表AIA_C17_LEAVE，并且把持久化后得到的JPA实例保存到流程变量中，这样可以方便后续节点更改实体的属性。

在流程中配置一个启动（start）类型监听器，其作用是读取用户填写的表单内容并创建一个LeaveJpaEntity（见代码清单17-3）对象并持久化到数据库。

代码清单17-3 请假实体

```
@Entity(name = "AIA_C17_LEAVE")
public class LeaveJpaEntity implements Serializable {
    private Long id;
    private String processInstanceId;
    private String userId;
```

```
private Date startTime;
private Date endTime;
private Date realityStartTime;
private Date realityEndTime;
private Date reportBackDate;
private Date applyTime;
private String leaveType;
private String reason;
private String deptLeaderApproved; // 部门领导是否同意
private String hrApproved; // 人事是否同意
// 省略了getter和setter
}
```

有了JPA实体还需要有相应的实体管理器（见代码清单17-4），由此可以看出这是一个被Spring管理的Bean对象。在代码清单17-4中，#1处注入了EntityManager对象，#2处创建一个新的JPA对象，#3处逐个获取变量并把值设置到创建的leave对象中，最后在#4处持久化实体到数据库。

代码清单17-4 请假实体管理器

```
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import java.util.Date;
@Service
public class LeaveEntityManager {
    @PersistenceContext #1
    private EntityManager entityManager;
    @Transactional
    public LeaveJpaEntity newLeave(DelegateExecution execution) {
        LeaveJpaEntity leave = new LeaveJpaEntity(); #2
        leave.setProcessInstanceId(execution.getProcessInstanceId()); #3-S
        leave.setUserId(execution.getVariable("applyUserId").toString());
        leave.setStartTime((Date) execution.getVariable("startTime"));
        leave.setEndTime((Date) execution.getVariable("endTime"));
        leave.setLeaveType(execution.getVariable("leaveType").toString());
        leave.setReason(execution.getVariable("reason").toString());
        leave.setApplyTime(new Date()); #3-E
        entityManager.persist(leave); #4
        return leave;
    }
}
```

接下来要在流程启动时调用LeaveEntityManager的新Leave（execution）方法（execution是引擎的内置变量），在Activiti Designer中添加一个流程启动监听器，如图17-3所示。

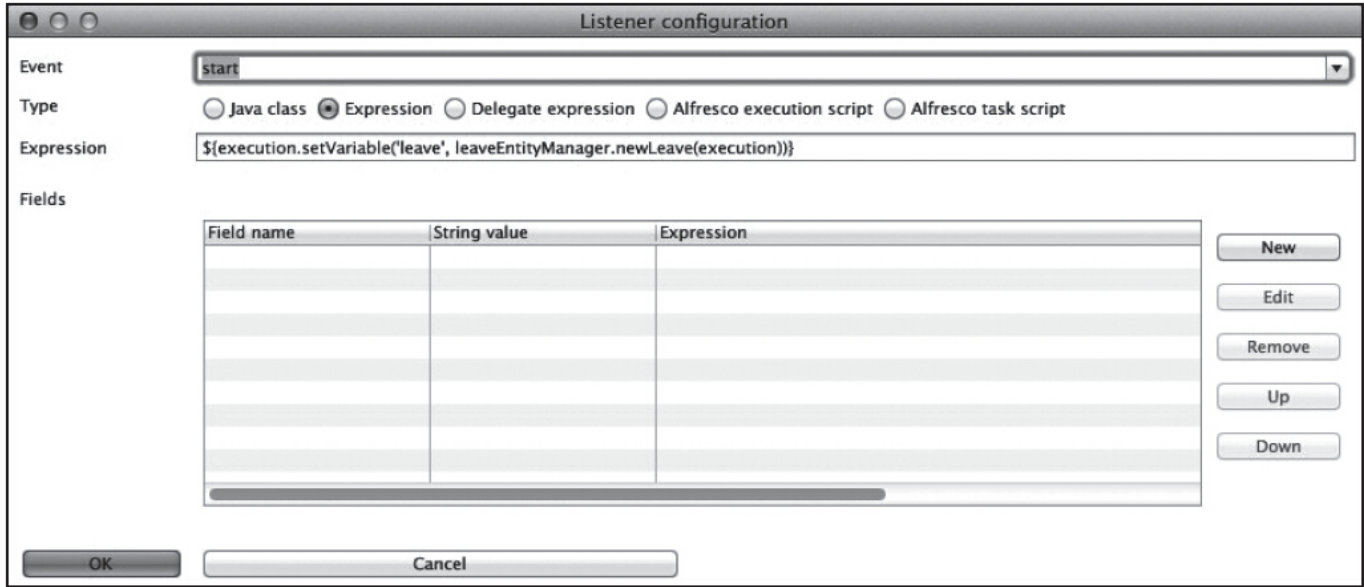


图17-3 配置流程启动监听器

对应的XML代码如下：

```
<process id="leave-jpa" name="请假流程-JPA" isExecutable="true">
  <extensionElements>
    <activiti:executionListener event="start" expression="${execution.setVariable('leave', leaveEntityManager.newLeave(execution))}">
    </activiti:executionListener>
  </extensionElements>
</process>
```

这样在流程启动后查询数据表AIA_C17_LEAVE就看到了一条数据，如图17-4所示。

记录 (AIA_C17_LEAVE)							
#	ID	APPLY_TIME	DEPT_LEADER_APPROVED	END_TIME	HR_APPROVED	LEAVE_TYPE	PROCESS_INSTANCE_ID
1	1	2014-04-24 00:20:00.337	true	2014-04-25 00:00:00.0		公休	5

REALITY_END_TIME	REALITY_START_TIME	REASON	REPORT_BACK_DATE	START_TIME	USER_ID
		公休2天		2014-04-24 00:00:00.0	kermit

图17-4 启动流程后插入的数据

同时观察一下Hibernate在控制台打印的SQL语句，其中有如下语句：

```
Hibernate:
insert into aia_c17_leave
(id, apply_time, dept_leader_approved, end_time, hr_approved, leave_type, process_instance_id, reality_end_time, reality_start_time, reason, report_back_date, start_time, user_id)
values (null, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
```

除了自定义表中的数据，我们再观察一下变量表名称为“leave”的变量，如图17-5所示。

记录 (ACT_RU_VARIABLE)												
#	ID_	REV_	TYPE_	NAME_	EXECUTION_ID_	PROC_INST_ID_	TASK_ID_	BYTEARRAY_ID_	DOUBLE_	LONG_	TEXT_	TEXT2_
1	6	1	string	applyUserId	5	5					kermit	
2	13	1	string	leaveType	5	5					公休	
3	14	1	date	startTime	5	5				1398268800000		
4	15	1	date	endTime	5	5				1398355200000		
5	16	1	string	reason	5	5					公休2天	
6	17	1	jpa-entity	leave	5	5					me.kafetiu.activiti.chapter17.entity.LeaveJpaEntity	1

图17-5 变量表数据

从图17-5中可以看出有6个不同类型的变量，其中最后一个变量的变量类型字段（TYPE_）为“jpa-entity”，该记录的TEXT_、TEXT2_字段分别记录了实体的完整类名及主键ID。

在引擎内部有一个专门处理JPA类型的变量类型类（org.activiti.engine.impl.variable.JPAEntityVariableType），当保存或读取变量时引擎会把JPA类型的记录交给这个类处理，当部门领导签收了任务并打开办理页面时就会读取任务相关的变量，此时观察一下控制台就可以看到有查询表AIA_C17_LEAVE的SQL语句输出，这说明每次读取变量时引擎会通过EntityManager接口获取实体数据。

17.2.2 更改JPA实体属性

上一小节通过在启动流程时执行表达式实现了创建数据记录（见图17-4），如果用户在办理时填写了任务表单，还需要把改动的数据更新到实体中，例如：

- 部门领导审核节点完成时保存审批意见。
- 人事审核节点完成时保存审批意见。
- 销假节点完成时保存员工填写的实际请假日期。

现在读者的脑海中应该是这样想的：“用户任务完成时无非就是该任务上添加一个complete类型的监听器而已”。

例如在“部门领导审核”节点要更新实体的属性，可以参照图17-6。

Listener configuration

Event

complete

Type

☐ Java class

☒ Expression

☐ Delegate expression

☐ Alfresco execution script

☐ Alfresco task script

Expression

\${leave.setDeptLeaderApproved(deptLeaderApproved)}

Fields

Field name	String value	Expression

New

Edit

Remove

Up

Down

OK

Cancel

图17-6 部门领导审核节点添加任务完成监听器

对应的XML如下：

```
<userTask id="deptLeaderAudit" name="部门领导审批" activiti:candidateGroups="deptLeader">
  <extensionElements>
    <activiti:taskListener event="complete" expression="${leave.setDeptLeaderApproved(deptLeaderApproved)}"></activiti:taskListener>
  </extensionElements>
</userTask>
```

此时以部门领导身份登录系统后会看到一条待办任务，签收并打开表单页面后如图17-7所示，在“审批意见”处选择“同意”，然后单击“完成任务”按钮，此时观察表AIA_C17_LEAVE的数据会发现字段“DEPT_LEADER_APPROVED”的值为“true”，表示部门领导节点的审批意见更新成功了。

任务内容

请假类型:

公休

请假开始日期:

2014-04-24

请假结束日期:

2014-04-25

请假原因:

公休2天

审批意见:

同意

✓完成任务

←返回列表

图17-7 部门领导审核任务表单

现在读者可能有疑问了，更改了属性后为什么没有调用EntityManager的持久化方法却能持久化对象到数据表？这是因为配置了事务管理器（参考本章源码的applicationContext.xml文件中名称为transactionManager的Bean对象），由Spring接管了引擎的事务，当然我们更改的实体也在被监管的范围内，所以在更改实体属性并提交事务时就自动执行了数据库的update操作。观察任务完成时控制台打印的SQL语句如下所示：

```
Hibernate:
update
  aia_c17_leave
set
  apply_time=?,
  dept_leader_approved=?,
  end_time=?,
  hr_approved=?,
  leave_type=?,
  process_instance_id=?,
  reality_end_time=?,
  reality_start_time=?,
  reason=?,
  report_back_date=?,
  start_time=?,
  user_id=?
where
  id=?
```

人事审批节点与部门领导审核节点类似，只是更改的属性不同而已，读者可以自行试验。下面给出了相应的XML配置，在任务完成后AIA_C17_LEAVE表的数据如图17-8所示，字段“HR_APPROVED” 值为 “true” 。

```
<userTask id="hrAudit" name="人事审批"
  activiti:candidateGroups="hr">
  <extensionElements>
    <activiti:taskListener event="complete" expression="${leave.setHrApproved      (hrApproved) }">
    </activiti:taskListener>
  </extensionElements>
</userTask>
```

记录 (AIA_C17_LEAVE)							
#	ID	APPLY_TIME	DEPT_LEADER_APPROVED	END_TIME	HR_APPROVED	LEAVE_TYPE	PROCESS_INSTANCE_ID
1	1	2014-04-24 00:20:00.337	true	2014-04-25 00:00:00.0	true	公休	5

图17-8 人事审批通过后数据库变化

当任务到达 “销假” 节点时，处理方式类似，把用户填写的实际请假日期更新到数据库，对应的XML如下，任务办理界面如图17-9所示，任务完成后数据库状态如图17-10所示。

```
<userTask id="reportBack" name="销假" activiti:assignee="${leave.userId}">
  <extensionElements>
    <activiti:taskListener event="complete" expression="${leave.setReportBackDate(reportBackDate) }"></activiti:taskListener>
    <activiti:taskListener event="complete" expression="${leave.setRealityStartTime(realityStartTime) }"></activiti:taskListener>
    <activiti:taskListener event="complete" expression="${leave.setRealityEndTime(realityEndTime) }"></activiti:taskListener>
  </extensionElements>
</userTask>
```

任务内容

请假类型:

公休

请假开始日期:

2014-04-24

请假结束日期:

2014-04-25

请假原因:

公休2天

(实际) 请假开始日期:

2014-04-24

(实际) 请假结束日期:

2014-04-25

销假日期:

2014-04-28

✔完成任务

◀返回列表

图17-9 销假节点任务表单

记录 (AIA_C17_LEAVE)							
#	ID	APPLY_TIME	DEPT_LEADER_APPROVED	END_TIME	HR_APPROVED	LEAVE_TYPE	PROCESS_INSTANCE_ID
1	1	2014-04-24 00:20:00.337	true	2014-04-25 00:00:00.0	true	公休	5

REALITY_END_TIME	REALITY_START_TIME	REASON	REPORT_BACK_DATE	START_TIME	USER_ID
2014-04-25 00:00:00.0	2014-04-24 00:00:00.0	公休2天	2014-04-28 00:00:00.0	2014-04-24 00:00:00.0	kermi

图17-10 销假任务完成后的数据

17.2.3 清理历史表单数据

到目前为止我们已经成功把数据存储方式从“行”转换成“列”，不过现在有一个问题需要解决，要把已结束流程的表单数据清理掉以减少数据量，从而提高引擎访问数据库的性能。可以在流程中配置一个流程完成监听器来清理历史表单数据，我们知道表单数据保存在ACT_HI_DETAIL表中，在流程结束时执行删除动作即可。

图17-11展示了添加的流程结束监听器，执行一个Delegate Expression。



图17-11 流程的结束监听器

对应的XML如下：

```
<activiti:executionListener event="end"
    delegateExpression="${leaveProcessEndListener}">
</activiti:executionListener>
```

名称为“leaveProcessEndListener”的Bean对象具体实现见代码清单17-5，其主要是行了一条SQL语句，从ACT_HI_DETAIL表中删除了和当前流程相关的数据

代码清单17-5 请假流程结束时

```
@Service
@Transactional
public class LeaveProcessEndListener implements ExecutionListener {
    @PersistenceContext
    private EntityManager entityManager;
    @Override
    public void notify(DelegateExecution execution) throws Exception {
        String processInstanceId = execution.getProcessInstanceId();
        String sql = "delete from act_hi_detail where proc_inst_id_ = ? and type_ = ?";
        int i = entityManager.createNativeQuery(sql)
            .setParameter(1, processInstanceId)
            .setParameter(2, "FormProperty")
            .executeUpdate();
        logger.debug("清理了 {} 条历史表单数据", i);
    }
}
```

17.3 本章小结

本章详细讲解了引擎与JPA的集成，可以把动态表单的流程数据保存到自定义的表中，这样可以把原本“无结构”的数据转换为“有结构”的数据，方便数据的查询、使用。

第18章 集成ESB

ESB (Enterprise Service Bus)，即企业服务总线，可用于集中管理企业内部应用对外的交互，提供对内部应用的协议转换、管理、监控、审计、分析等功能。除了对单个服务的管理功能外，ESB还可以把企业内部各个分散的功能封装成一个整体服务，这对于服务调用者来说只需要关注一个接口即可。

目前比较流行的ESB平台有Camel、Mule、ServiceMix、Spring Integration等，这些都实现了EIP（Enterprise Integration Patterns，企业整合模式）。本章将针对Camel、Mule分别介绍如何与Activiti集成。

18.1 Camel简介与快速入门

Camel是由Apache开源组织使用Java语言开发的一个强大的基于规则的路由及媒介引擎，采用URI的方式来描述多个需要整合的组件。开发人员可以使用DSL（Domain Specific Language）配置路由规则。下面的代码就是一个简单的路由示例，其功能是把/tmp/folder1目录的文件转移到/tmp/folder2目录。

```
from("file:/tmp/folder1").to("file: /tmp/folder2");
```

有关Camel的资料可以访问官方网站<http://camel.apache.org>来了解。

本节介绍一个基于Camel的简单例子——HelloCamel，以便快速入门。熟悉Camel的读者可以跳过本节直接学习下一小节。

 **提示** 本章的相关代码可以在本书源代码chapter18-esb模块的chapter18目录中找到。

使用Camel需要先在Maven的pom.xml文件中添加依赖定义，代码如下：

```
<dependency>
<groupId>org.apache.camel</groupId>
<artifactId>camel-spring</artifactId>
<version>2.11.4</version>
</dependency>
```

Camel的核心是路由（Route）。路由可以用XML的方式配置，也可以自定义一个继承自RouteBuilder接口的Java类注册到CamelContext对象中。本书中的Camel路由均使用Java类的方式定义。

在代码清单18-1中定义的路由用于判断请假天数是否超过3天，并把结果设置到名称为“leave”的Bean对象中。Bean的定义见代码清单18-2，针对代码清单18-1的路由功能的验证见代码清单18-3。

代码清单18-1 Camel路由实现的请假逻辑

```
public class CamelLeaveRoute extends RouteBuilder {
    @Override
    public void configure() throws Exception {
        from("direct:start") // 路由URI
        .log(LoggingLevel.INFO, "接收到消息: ${in.body}")
        .choice()
        .when(simple("${in.body[days]} > 3")) // 路由条件
        .log("用户${body[userId]} 请假天数超过 3 天")
        .beanRef("leave", "setResult(true)") // 调用Bean的setResult方法
        .endChoice();
    }
}
```

代码清单18-2 CamelLeaveBean类

```
public class CamelLeaveBean implements Serializable {
    private Boolean result = false;
    // 省略getter和setter
}
```

代码清单18-3 验证Camel路由实现的请假逻辑

```
public class CamelTest {
    @Test
    public void testCamelForBean() throws Exception {
        JndiContext context = new JndiContext();
        CamelLeaveBean leave = new CamelLeaveBean();
        context.bind("leave", leave); // 把Bean对象绑定到CamelContext中
        CamelContext camelContext = new DefaultCamelContext(context);
        camelContext.addRoutes(new CamelLeaveRoute()); // 把路由注册到CamelContext
        camelContext.start();
        ProducerTemplate tpl = camelContext.createProducerTemplate();
        // true
        tpl.sendBody("direct:start", Collections.singletonMap("days", 2)); // #1-S
        assertFalse(leave.getResult()); // #1-E
        // false
        tpl.sendBody("direct:start", Collections.singletonMap("days ", 5)); // #2-S
        assertTrue(leave.getResult()); // #2-E
        camelContext.stop();
    }
}
```

代码清单18-3中的#1-S、#2-S处分别发送了请假2天、5天的路由请求，运行单元测试后，#1-E、#2-E均验证通过，这说明路由（见代码清单18-1）可以提供正确的功能。

18.2 在流程中调用Camel

在本书开头就强调过，Activiti不仅是一个流程引擎，确切地说更是一个BPM平台。例如，图18-1的流程图中集成了Camel任务来调用Camel路由从而影响流程的走向，该流程的作用是：申请的假期天数大于3天由部门经理审批，否则由直接领导审批。

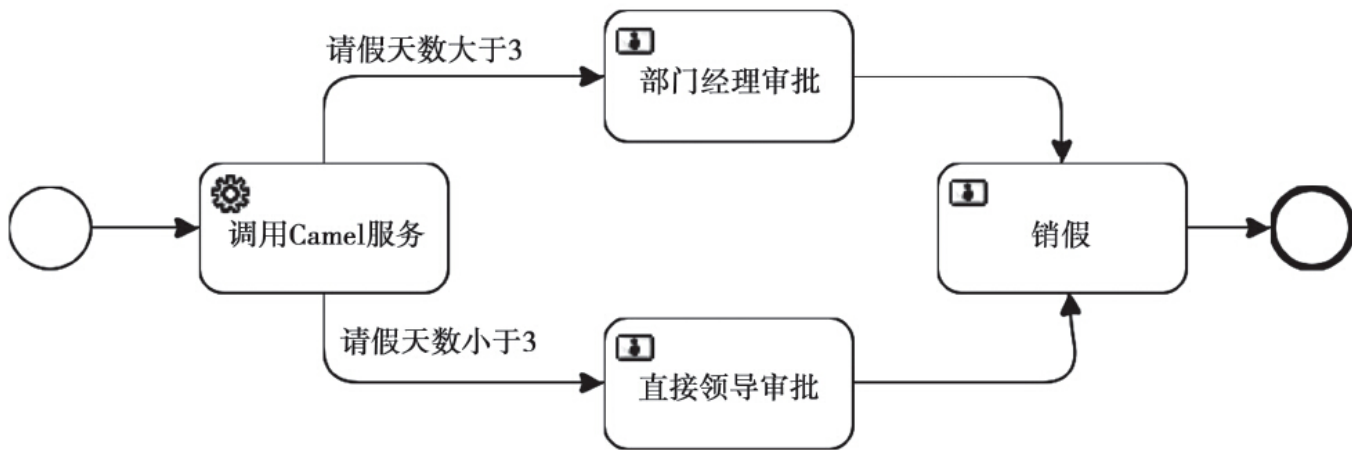


图18-1 集成了Camel任务的流程图

BPMN 2.0规范中没有单独的Camel任务，在第4章中介绍Camel任务时就说明了Camel任务是基于ServiceTask进行扩展的。一个Camel任务的XML定义如下：

```
<serviceTask id="invokeCamel" name="Camel Task"
  activiti:type="camel"></serviceTask>
```

“请假天数大于3天” 的输出流条件为`$(deptLeaderAudit)`，“请假天数小于3天” 的输出流条件为`$(! deptLeaderAudit)`，相关XML如下：

```
<sequenceFlow id="flow2" name="请假天数大于3天" sourceRef="invokeCamel" targetRef="deptLeaderAudit">
  <conditionExpression xsi:type="tFormalExpression">
    <![CDATA[${deptLeaderAudit}]]>
  </conditionExpression>
</sequenceFlow>
<userTask id="directlyLeaderAudit" name="直接领导审批"/>
<sequenceFlow id="flow3" name="请假天数小于3天" sourceRef="invokeCamel" targetRef="directlyLeaderAudit">
  <conditionExpression xsi:type="tFormalExpression">
    <![CDATA[${!deptLeaderAudit}]]>
  </conditionExpression>
</sequenceFlow>
```

从条件可以看出，需要在运行时向流程实例提供“deptLeaderAudit”变量，确切地说应该是由“调用Camel服务”这个任务提供，这也就是我们学习的目标。

18.2.1 Camel依赖及配置

在本书源代码chapter18-esb/src/test/resources/chapter18/applicationContext.xml文件中用Spring代理了Activiti引擎，其中除了定义和Activiti有关的必要配置之外，还添加了camel-spring模块提供的自动扫描功能，配置如下：

```
<camelContext id="camelContext"
  xmlns="http://camel.apache.org/schema/spring">
  <packageScan>
    <package>
      me.kafeitu.activiti.chapter18.esb.camel.activiti
    </package>
  </packageScan>
</camelContext>
```

以上配置可以在初始化CamelContext时把包me.kafeitu.activiti.chapter18.esb.camel.activiti中的路由类（继承RouteBuilder类）注册到CamelContext对象中。

另外还需要添加Activiti的Camel模块，其Maven依赖定义如下：

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-spring</artifactId>
  <version>2.11.4</version>
</dependency>
<dependency>
  <groupId>org.activiti</groupId>
  <artifactId>activiti-camel</artifactId>
  <version>${activiti.version}</version>
</dependency>
```

注意 在使用Activiti 5.15与camel-spring 2.12.x整合时会遇到Camel内部对象org.apache.camel.impl.DefaultMessageHistory不能序列化的问题，所以本书使用的是Camel 2.11.x，未来Activiti的Camel模块应该会修复此问题。

18.2.2 定义Camel路由

activiti-camel模块提供了Camel与Activiti通信的桥梁，由Activiti触发Camel任务后执行相符的路由，同时可以选择把流程相关的变量传递给路由，在路由处理结束后还可以有选择性地把路由产生的结果以变量的形式回传给流程实例。

activiti-camel模块定义了“activiti”类型的路由URI协议，一个典型的路由格式如下，表18-1列出了对URI的解释。

```
from("activiti://leaveWithCamel:invokeCamel?fooVar=bar")
  .to("log: catch route days: ${property.days}")
```

表18-1 activiti-camel模块提供的URI协议

属 性 名 称	属 性 说 明
activiti://	协议开头
leaveWithCamel	流程定义的 KEY
invokeCamel	流程中 Camel 服务的任务 ID
fooVar=bar	路由 URI 的参数

代码清单18-4列出了和流程结合的路由配置相关代码，稍后会介绍其他几种定义方式。

代码清单18-4 CamelWithActivitiLeaveRoute.java

```
public class CamelWithActivitiLeaveRoute extends RouteBuilder {
    public void configure() throws Exception {
        from("activiti://leaveWithCamel:invokeCamel")                #1
        .log(LoggingLevel.INFO, "接收到消息: ${property.days}")
        .log(LoggingLevel.INFO, "Camel路由中接收到流程ID:
            ${property." + ActivitiProducer.PROCESS_ID_PROPERTY + "}")
        .choice().when(simple("${property.days} > 3"))                #2
        .setProperty("deptLeaderAudit",simple("true", Boolean.class))    #3
        .otherwise().setProperty("deptLeaderAudit", simple("false", Boolean.class));    #4
        .endChoice().setBody().properties();                            #5
    }
}
```

代码清单18-4的#1处声明了路由的URI，#2处使用流程变量作为条件判断的依据，#3、#4处设置变量“deptLeaderAudit”的值，#5处路由中设置的属性作为响应Body对象的值，这样在流程中就可以使用在路由中设置的属性，这一切都由Activiti的Camel模块来完成。

18.2.3 执行单元测试

一切都准备好了，现在可以验证Activiti与Camel的整合是否可以提供我们预期的功能了。代码清单18-5列出了配套的单元测试来帮我们验证流程的执行结果。

代码清单18-5 Activiti与Camel整合的单元测试类ActivitiWithCamelTest

```
@ContextConfiguration("classpath:chapter18/applicationContext.xml")
public class ActivitiWithCamelTest extends SpringActivitiTestCase {
    @Deployment(resources = {"chapter18/leaveWithCamel.bpmn"})
    public void testCamel() throws Exception {
        Map<String, Object> vars = new HashMap<String, Object>();
        vars.put("days", 5); // 设置请假天数                #1
        ProcessInstance processInstance = runtimeService
            .startProcessInstanceByKey("leaveWithCamel", vars);
        assertNotNull(processInstance);
        Task task = taskService.createTaskQuery().singleResult();
        assertEquals("部门经理审批", task.getName());
    }
}
```

18.2.4 URI输入参数

在代码清单18-4中，#1处的from()方法的参数是一个标准的URI，如果要干预Camel组件的功能，需要传递一些参数，本节将——介绍路由URI中支持的参数，包括输入参数和输出参数两类。完整代码可参考本书源代码文件chapter18-esb/src/main/java/xxx/CamelWithActivitiLeaveRoute.java在该文件中启用一种参数配置方式后运行代码清单18-5所示的测试代码均能通过单元测试。

1.copyVariablesFromProperties

默认属性，该属性可以把Activiti变量复制成Camel属性，在路由中可以使用像\${property.foo}这样的表达式获取该属性的值，与代码清单18-4中的setBody().properties()功能等价。

```
from("activiti://leaveWithCamel:invokeCamel
    ?copyVariablesFromProperties=true")
.choice()
    .when(simple("${property.days} > 3"))
    .setProperty("deptLeaderAudit",
        simple("true", Boolean.class))
    .otherwise()
    .setProperty("deptLeaderAudit",
        simple("false", Boolean.class));
```

除了可以使用URI参数的形式启用该功能外，还可以在流程文件中配置一个Camel行为实现类，效果与参数功能相同。

```
<serviceTask id="serviceTask1" activiti:type="camel">
    <extensionElements>
        <activiti:field name="camelBehaviorClass"
            stringValue="org.activiti.camel.impl.
                CamelBehaviorCamelBodyImpl" />
    </extensionElements>
</serviceTask>
```

2.copyCamelBodyToBody

只把名称为“camelBody”的Activiti变量复制成Camel消息体，如果camelBody的值是Map对象，在路由中可以通过\${body[foo]}获取属性的值，如果是纯字符，可以使用\${body}获取。

```
from("activiti://leaveWithCamel:invokeCamel?copyCamelBodyToBody=true&copyVariablesFromProperties=true")
.choice().when(simple("${body[days]} > 3"))
    .setProperty("deptLeaderAudit",
        simple("true", Boolean.class))
    .otherwise().setProperty("deptLeaderAudit",
        simple("false", Boolean.class));
```

需要特别说明一点，要在流程中传递一个名称为“camelBody”的变量，否则路由获取不到body的属性，把代码清单18-5中的#1处替换成如下代码：

```
vars.put("camelBody", Collections.singletonMap("days", 5));
```

除了使用URI参数的形式启用该功能外，还可以在流程文件中配置一个Camel行为实现类，效果与参数功能相同。

```
<serviceTask id="serviceTask1" activiti:type="camel">
  <extensionElements>
    <activiti:field name="camelBehaviorClass"
      stringValue="org.activiti.camel.impl.
        CamelBehaviorCamelBodyImpl" />
  </extensionElements>
</serviceTask>
```

3.copyVariablesToBodyAsMap

与属性`copyCamelBodyToBody`功能类似，但是这个属性开启后会把Activiti所有变量赋值到一个Map对象中作为Camel的消息体。

```
from("activiti://leaveWithCamel:invokeCamel?copyVariablesToBodyAsMap=true&copyVariablesFromProperties=true")
  .choice().when(simple("${body[days]} > 3"))
    .setProperty("deptLeaderAudit",
      simple("true", Boolean.class))
    .otherwise().setProperty("deptLeaderAudit",
      simple("false", Boolean.class));
```

18.2.5 URI输出参数

1.默认配置

如果Camel消息体是一个Map对象，那么路由执行结束后会把Camel的每一个属性复制成Activiti流程实例变量，否则把整个Camel消息体作为Activiti变量“camelBody”的值。

2.copyVariablesFromProperties

该属性可以将Camel属性以相同的名称复制成Activiti变量。

```
from("activiti://leaveWithCamel:invokeCamel?copyVariablesFromProperties=true")
  .choice().when(simple("${property.days} > 3"))
    .setProperty("deptLeaderAudit",
      simple("true", Boolean.class))
    .otherwise().setProperty("deptLeaderAudit",
      simple("false", Boolean.class));
```

3.copyVariablesFromHeader

此属性的功能和`copyVariablesFromProperties`类似，不同的是此属性是从Header读取而不是从Properties中读取。

```
from("activiti://leaveWithCamel:invokeCamel?copyVariablesFromProperties=true")
  .choice().when(simple("${property.days} > 3"))
    .setHeader("deptLeaderAudit",
      simple("true", Boolean.class))
    .otherwise().setHeader("deptLeaderAudit",
      simple("false", Boolean.class));
```

4.copyCamelBodyToBodyAsString

`CopyCamelBodyToBodyAsString`的功能和默认配置的功能相同，但是当Camel消息体不是Map对象时先把消息转换成字符串再设置到“camelBody”变量中。

18.3 异步Camel任务

如果Camel任务执行时间比较长，可以把异步功能打开，并且在Camel任务后面添加一个Receive Task接收Camel任务执行完成的通知节点。图18-2展示了一个异步Camel任务。



图18-2 异步Camel任务流程图

图18-2所示流程对应的XML如下：

```
<serviceTask id="asyncCamelTask" name="Camel任务"
  activiti:async="true"></serviceTask>
<receiveTask id="receiveAsyncCamel"
  name="等待Camel任务完成"></receiveTask>
<sequenceFlow id="flow2" sourceRef="asyncCamelTask"
  targetRef="receivetask1"></sequenceFlow>
```

当执行到Camel任务时流程实例会等待接收信号（Signal），可以通过下面的代码由Camel向Activiti引擎发送信号。

```
from("activiti:camelAsyncProcess:asyncCamelTask")
  .to("activiti: camelAsyncProcess:receiveAsyncCamel ");
```

也可以使用Activiti的标准API发送信号恢复流程的执行状态：

```
runtimeService.signal(executionId);
```

18.4 在Camel中启动流程

通过以上内容我们学习了如何整合Camel与Activiti，以及两者如何通信，但是这些都是建立在流程实例已经启动的状态下。本节将介绍如何在Camel中启动Activiti流程，如代码清单18-6所示。

代码清单18-6 在Camel中启动Activiti流程的路由类

```
public class StartProcessByCamelRoute extends RouteBuilder {
    @Override
    public void configure() throws Exception {
        from("activiti://masterProcess:startSubProcess")
            .to("activiti:camelSubProcess");
        // 通过Camel启动流程URI只需要传递两个参数即可
    }
}
```

图18-3展示了两个流程，分别为masterProcess（主流程）和camelSubProcess（子流程），对应的单元测试如下：

```
@Deployment(resources = {
    "chapter18/masterProcess.bpmn",
    "chapter18/camelSubProcess.bpmn"})
public void testStartProcessByCamel() throws Exception {
    ProcessInstance processInstance = runtimeService
        .startProcessInstanceByKey("masterProcess");
    assertNotNull(processInstance);
    // 检查子流程是否已启动
    ProcessInstance subProcess = runtimeService
        .createProcessInstanceQuery()
        .processDefinitionKey("camelSubProcess")
        .singleResult();
    assertNotNull(subProcess);
}
```

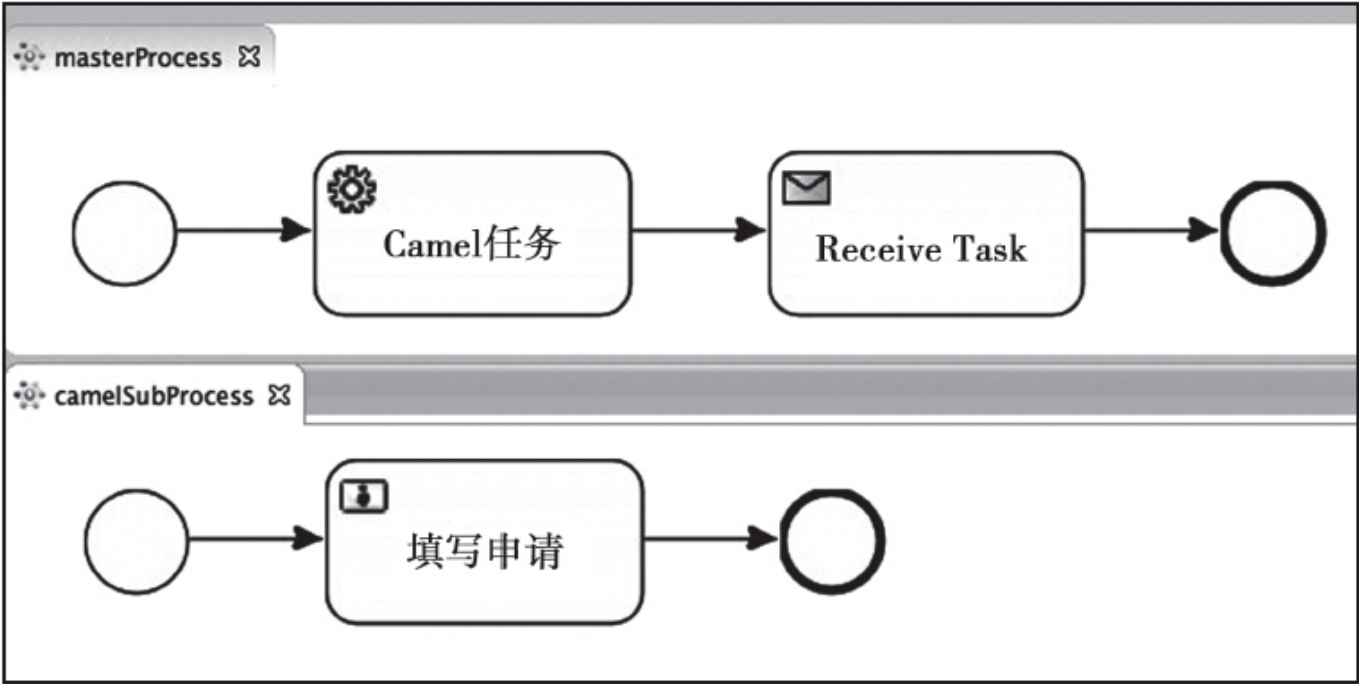


图18-3 在Camel中启动流程的相关流程定义

18.5 集成Mule

Mule和Camel类似，都是轻量级的ESB平台实现，提供了一个以EIP为基础的框架，可以把各种协议、第三方框架的实现等方便地集成到Mule中，Mule团队也针对Activiti提供了一个专门的模块用于建立两者的通信桥梁。

本节 Maven 依赖配置可以从chapter18-esb模块的pom.xml文件中查看，由于配置较多这里不再列举。Mule版本号：3.3.0；Spring版本号：3.2.7.RELEASE；Mule-Activiti组件版本号：3.2.0。

18.5.1 Mule快速入门

与讲解Camel类似，为了照顾不熟悉Mule的读者，这里先介绍一些基础知识，让大家快速入门，然后再介绍Activiti与Mule的集成。熟悉Mule的读者可以跳过本节。

Camel中的路由在Mule中称为Flow，这些Flow定义在一个XML文件中，可以定义Flow的访问路径、调用的服务以及请求内容等信息。代码清单18-7列出了入门示例的XML配置，该文件为chapter18-esb/src/test/resources/chapter18/mule/mule-standalone-config.xml。

代码清单18-7 mule-standalone-config.xml

```
<mule xmlns="http://www.mulesoft.org/schema/mule/core"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:vm="http://www.mulesoft.org/schema/mule/vm"
      xsi:schemaLocation="http://www.mulesoft.org/schema/mule/core
        http://www.mulesoft.org/schema/mule/core/3.1/mule.xsd
        http://www.mulesoft.org/schema/mule/vm
        http://www.mulesoft.org/schema/mule/vm/3.1/mule-vm.xsd">
  <flow name="HelloMule">
    <vm:inbound-endpoint path="helloMule" exchange-pattern="request-response" />
    <component class="me.kafeitu.activiti.chapter18.esb.mule.LeaveMule Service">
      <method-entry-point-resolver>
        <include-entry-point method="isDeptLeaderApproved"/>
      </method-entry-point-resolver>
    </component>
  </flow>
</mule>
```



```
        </method-entry-point-resolver>
    </component>
</flow>
</mule>
```

代码清单18-7是一个比较简单的Mule配置文件，XML头部要引入必要的Scheme：#1处的vm是Mule的一种协议，用参数“path”定义访问路径，这样我们通过vm://helloMule就可以访问到该服务；#2处定义了一个组件并指定了class属性，表示当访问该Flow时会执行配置的类文件并调用#3处定义的“isDeptLeaderApproved”方法。

代码清单18-7中#2处的Java文件定义了一个简单的服务，见代码清单18-8所示，LeaveMuleService类只定义了一个方法，接收一个名称为days的参数，然后返回与3比较大小后的结果。

代码清单18-8 LeaveMuleService.java

```
public class LeaveMuleService {
    public boolean isDeptLeaderApproved(Integer days) {
        return days > 3;
    }
}
```

定义了XML配置文件及服务类，现在我们可以来检验一下这个简单的Mule服务能否提供预期的结果。

和Camle类似，Mule也需要先创建上下文对象（Context）才能调用Mule公开的API。代码清单18-9列出了一个JUnit单元测试类用于验证Mule服务的执行结果。

代码清单18-9 Mule入门单元测试类HelloMuleTest.java

```
public class HelloMuleTest {
    @Test
    public void testMule() throws Exception {
        MuleContext muleContext = new DefaultMuleContextFactory() // 初始化Context
            .createMuleContext("chapter18/mule/mule-standlone-config.xml");
        muleContext.start();
        MuleClient muleClient = new DefaultLocalMuleClient(muleContext);
        Integer days = 5; // 定义请假天数，结果应该返回true
        MuleMessage message = muleClient.send("vm://helloMule", // 服务协议URI
            new DefaultMuleMessage(days, muleContext));
        assertNotNull(message.getPayload());
        Boolean result = (Boolean) message.getPayload();
        assertTrue(result);
    }
}
```

单元测试执行通过说明定义的Mule服务可以正常提供功能，接下来继续学习如何在Activiti流程中调用Mule服务。

18.5.2 Mule、Spring与Activiti集成

在开始介绍这一小节之前，需要先了解一下Mule与Spring如何集成。因为示例代码中使用Spring代理Activiti引擎（这也是大多数开发人员采用的方式）。Mule与Activiti均提供了Spring模块，所以我们使用Spring作为两者的黏合剂。

Mule与Spring都可以作为Bean容器使用，如果系统环境以Mule为主，可以把Mule作为主容器，然后引入（import）Spring的bean对象；如果系统环境以Spring为主，可以把Spring作为主容器，在创建MuleContext对象时引入Mule的配置文件。选择哪种配置方式可以根据开发环境决定。

下面列出了chapter18-esb/src/test/resources目录的树状结构，根据选择的主容器划分了目录，以便读者理解参考。

```
├── mule-master
│   ├── mule-master.xml           // Mule作为主容器
│   └── spring-activiti-lesser.xml // Spring配置被加载（import）
└── spring-master
    ├── mule-lesser.xml           // Spring作为主容器
    └── spring-master.xml         // Mule配置被加载（import）
```

1.以Mule为主容器

把Mule作为主容器时可以参考代码清单18-7所示的配置代码，可以在代码清单18-7的头部添加以下配置把Spring配置文件的代码引入（import）Mule配置文件中（mule-master.xml）。

```
<spring:beans>
  <spring:import resource="spring-activiti-lesser.xml" />
</spring:beans>
```

要在Mule中定义使用Activiti组件，需要在XML头部导入mule-activiti模块的Scheme定义，XSD配置如下：

```
<mule xmlns:activiti=http://www.mulesoft.org/schema/mule/activiti-embedded
xsi:schemaLocation=http://www.mulesoft.org/schema/mule/activiti-embedded http://www.mulesoft.org/schema/mule/activiti-embedded/3.2/mule-activiti-embedded.xsd>
...
<activiti:connector name="actServer"
  repositoryService-ref="repositoryService"
  runtimeService-ref="runtimeService"
  taskService-ref="taskService"
  historyService-ref="historyService" />
<flow name="StartLeaveProcessUseMule">
  <vm:inbound-endpoint path="startLeaveProcess"
    exchange-pattern="request-response" />
  <activiti:create-process parametersExpression="
    #[header:INBOUND:createProcessParameters]" />
</flow>
...
</mule>
```

在以上配置中，“activiti:connector”的作用是把Spring代理的Activiti Service接口注入Mule容器，以便mule-activiti模块使用；在名称为“StartLeaveProcessUseMule”的Flow中配置 inbound-endpoint（path属性为“startLeaveProcess”）中定义了“activiti:create-process”组件，用来调用mule-activiti组件提供的启动流程功能（示例见代码清单18-11）。

在Spring配置文件（mule-master/spring-activiti-lesser.xml）中需要为代理的Activiti引擎注入muleContext对象，这样在Activiti流程中才可以执行Mule任务，相关配置如下：

```
<bean id="processEngineConfiguration" class="org.activiti.spring.SpringProcessEngineConfiguration">
  ... // 省略了其他属性
  <property name="beans">
    <map>
      <entry key="muleContext"
        value-ref="_muleContext"/>
    </map>
  </property>
</bean>
```

在以上代码中，属性 “_muleContext” 是一个Bean的引用，在初始化MuleContext时创建，然后注入并共享给Spring容器。在Activiti引擎内调用Mule任务时必须提供名称为 “muleContext” 的MuleContext对象。

2.以Spring为主容器

把Spring作为主容器时需要由Spring代理MuleContext，也就是由Spring触发Mule-Context对象的初始化方法，在Spring配置文件（spring-master.xml）中加入以下配置即可。

```
<!-- MuleContext工厂 -->
<bean id="muleFactory" class="org.mule.context.DefaultMuleContextFactory"></bean>
<bean id=" muleContext" factory-bean="muleFactory"
    factory-method="createMuleContext">
    <constructor-arg type="java.lang.String"
        value="mule/spring-master/mule-lesser.xml" />
</bean>
```

此时Mule的配置（mule/spring-master/mule-lesser.xml）被引入Spring配置文件中，所以只需要定义具体的服务即可。

18.5.3 在流程中调用Mule

图18-4展示了包含Mule任务的流程图，与讲解Camel时的流程图类似。代码清单18-10列出了图18-4中与Mule任务（调用Mule服务）相关的XML配置代码。

代码清单18-10 图18-4中与Mule任务相关的XML配置

```
<serviceTask id="invokeMule" name="调用Mule服务" activiti:type="mule"> //定义任务类型
<extensionElements>
<activiti:field name="endpointUrl"> // 定义调用Mule服务的URI
<activiti:string><![CDATA[vm://checkDays]]></activiti:string>
</activiti:field>
<activiti:field name="language"> // 设置使用参数解析语言，使用引擎默认的JUEL
<activiti:string><![CDATA[juel]]></activiti:string>
</activiti:field>
<activiti:field name="payloadExpression"> // 请求Mule服务的表达式
<activiti:expression>${days}</activiti:expression>
</activiti:field>
<activiti:field name="resultVariable"> //设置Mule服务执行完成后结果保存到哪个变量中
<activiti:string><![CDATA[deptLeaderAudit]]></activiti:string>
</activiti:field>
</extensionElements>
</serviceTask>
```

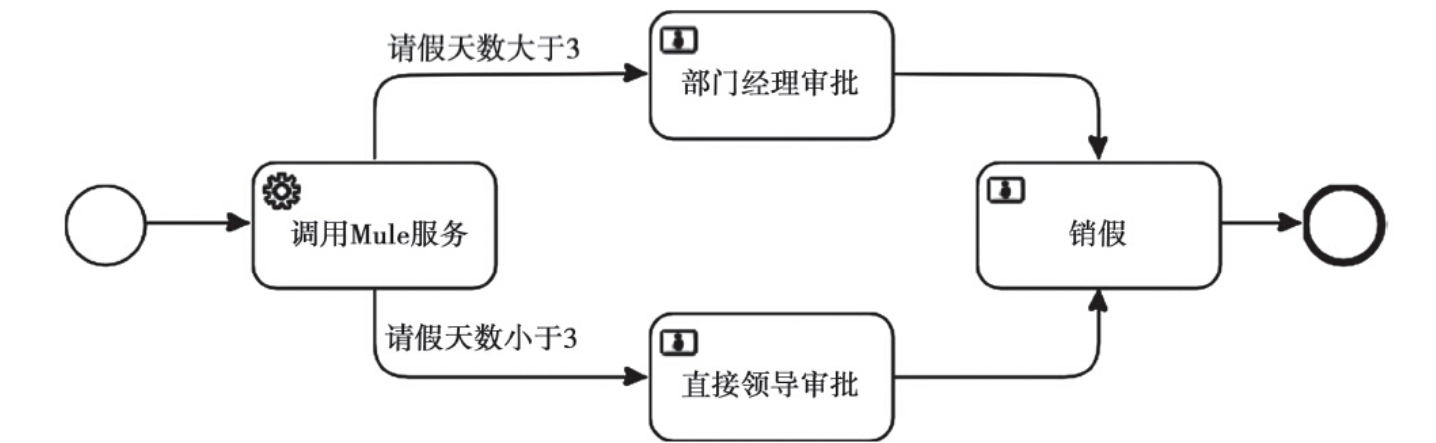


图18-4 包含Mule任务的流程图

1.以Mule为主容器

代码清单18-11列出了把Mule作为主容器时的测试代码。

代码清单18-11 测试在Activiti流程中调用Mule服务——Mule作为主容器

```
@Test
public void testMuleMaster() throws Exception {
    MuleContext muleContext = new DefaultMuleContextFactory() // 初始化MuleContext
        .createMuleContext("mule/mule-master/mule-master.xml");
    muleContext.start(); // 启动Mule
    MuleRegistry registry = muleContext.getRegistry();
    RepositoryService repositoryService = // 从Mule容器获取Activiti Service接口
        (RepositoryService) registry.get("repositoryService");
    repositoryService.createDeployment() // 部署流程
        .addClasspathResource("mule/leaveWithMule.bpmn").deploy();
    MuleClient muleClient = new DefaultLocalMuleClient(muleContext);
    DefaultMuleMessage message = new DefaultMuleMessage("", muleContext);
    Map<String, Object> variableMap = new HashMap<String, Object>();
    variableMap.put("days", 5); // 设置days变量为5天
    // 由Mule启动流程需要传递流程定义KEY参数
    variableMap.put("processDefinitionKey", "leaveWithMule");
    message.setProperty("createProcessParameters", // 调用Mule服务时的参数
        variableMap, PropertyScope.OUTBOUND);
    MuleMessage responseMessage = // 向Mule发送启动流程服务
        muleClient.send("vm://startLeaveProcess", message);
    ProcessInstance processInstance = (ProcessInstance) responseMessage.getPayload();
    TaskService taskService = registry.get("taskService");
    Task task = taskService.createTaskQuery().singleResult();
    // 测试通过说明Mule服务被调用，并设置了deptLeaderAudit变量为true
    assertEquals("部门经理审批", task.getName());
    muleContext.stop();
    muleContext.dispose();
}
```

2.以Spring为主容器

代码清单18-12列出了把Spring作为主容器时的测试代码。

代码清单18-12 测试在Activiti流程中调用Mule服务——Spring作为主容器

```
@Test
public void testSpringMaster() throws Exception {
    ApplicationContext ctx = // 初始化Spring容器
        new ClassPathXmlApplicationContext("mule/spring-master/spring-master.xml");
    MuleContext mc = (MuleContext) ctx.getBean("muleContext");
    mc.start(); // 从Spring获取MuleContext并开启服务
    RepositoryService repositoryService = // 从Spring获取Activiti Service
        (RepositoryService) ctx.getBean("repositoryService");
    repositoryService.createDeployment()
        .addClasspathResource("mule/leaveWithMule.bpmn").deploy();
    RuntimeService runtimeService = (RuntimeService) ctx.getBean("runtimeService");
    Map<String, Object> variableMap = new HashMap<String, Object>();
    variableMap.put("days", 5);
    runtimeService.startProcessInstanceByKey("leaveWithMule", variableMap);
    TaskService taskService = (TaskService) ctx.getBean("taskService");
    Task task = taskService.createTaskQuery().singleResult();
    assertEquals("部门经理审批", task.getName()); // 验证结果
    mc.stop();
}
```

18.6 本章小结

本章主要针对目前流行且Activiti支持的两个ESB平台——Camel和Mule做了详细的介绍，并针对不熟悉这两个平台的读者提供了快速入门示例，在此基础上——讲解了如何在Activiti流程中调用Camel路由和Mule服务。

在Activiti中集成Camel时可以通过URI的方式配置参数传递，在路由执行完成后将结果返回给引擎的方式，本章都——做了讲解；在介绍Mule时考虑到读者面对的开发环境不同（以Mule为主容器或以Spring为主容器），还介绍了两种配置方式的不同。

第19章 统一身份管理

在一个完整的系统中，身份模块是必不可少的，用于管理系统的用户、组织、角色或者岗位。在第5章中我们详细介绍过Activiti的用户与组，以及如何通过引擎提供的API维护两者之间的关系。国内大多数系统都会有组织的管理，而在Activiti中并没有组织的概念，也没有角色（Role）的管理，取而代之的是组（Group）。

现在遇到的问题就是业务系统有一套用户管理系统，而Activiti也提供了一套用户管理的机制（虽然相对来说比较简单），同时维护两套用户数据从成本和易用性上考虑都不太可取，本章将介绍几种用户数据同步的方案。

19.1 一套典型的身份系统

图19-1展示的是一个典型的身份系统ER图，相比Activiti的身份模块（以ACT_ID开头的表）多出了几张表，例如组织结构表（ID_ORG）、资源表（ID_RESOURCES）。图19-1的设计代表了大多数的设计思想，也基本能满足国内客户对身份管理的需求，本章内容将以此作为背景资料进行讲解。

针对图19-1的ER图定义了相应的实体，如图19-2所示，包含了以下几个实体对象（为了简便起见只定义了必要的几个属性）：

- AiaUser：用户实体。
- AiaDepartment：部门实体。
- AiaResource：资源实体。
- AiaRole：角色实体。

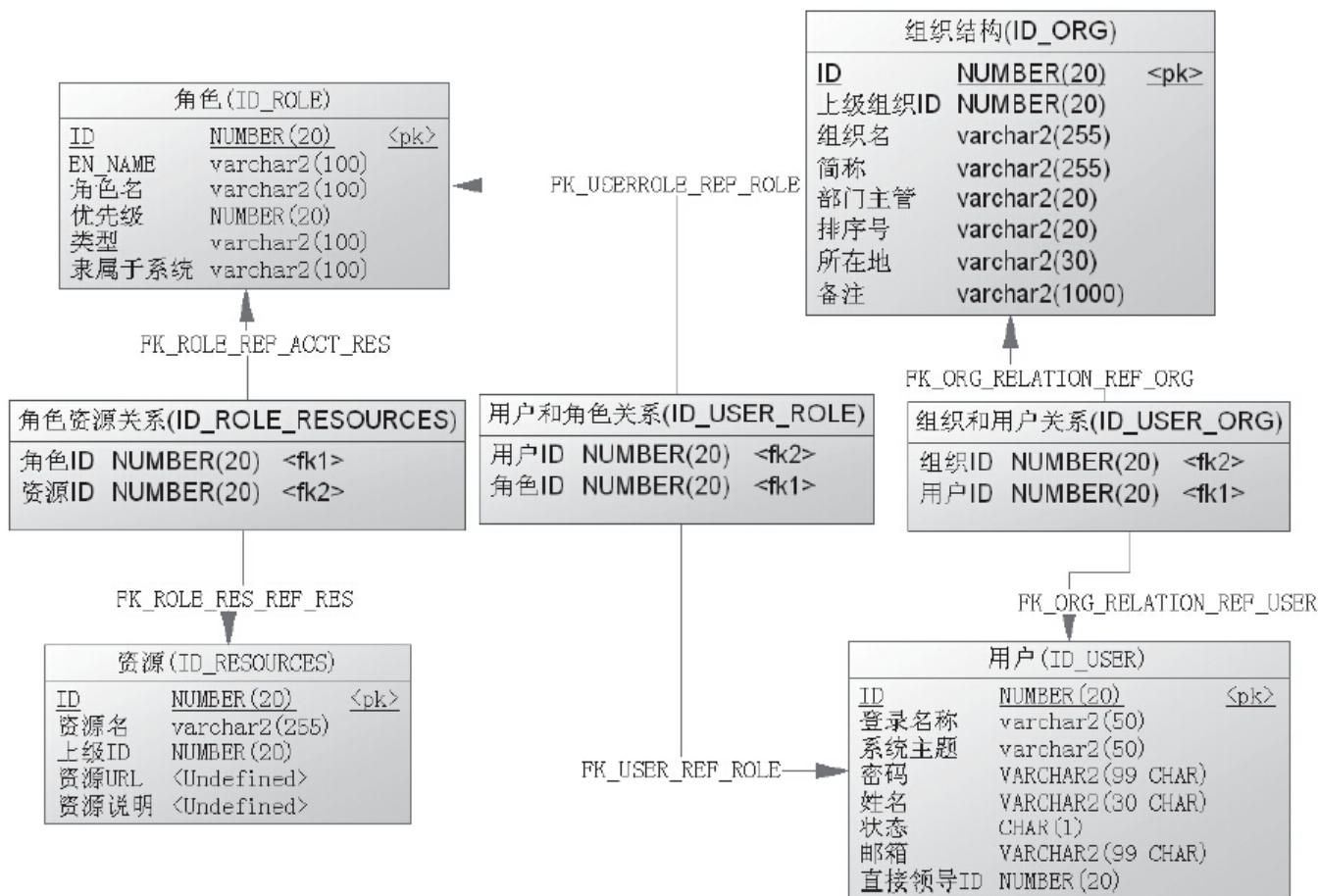


图19-1 一个典型的身份系统ER图

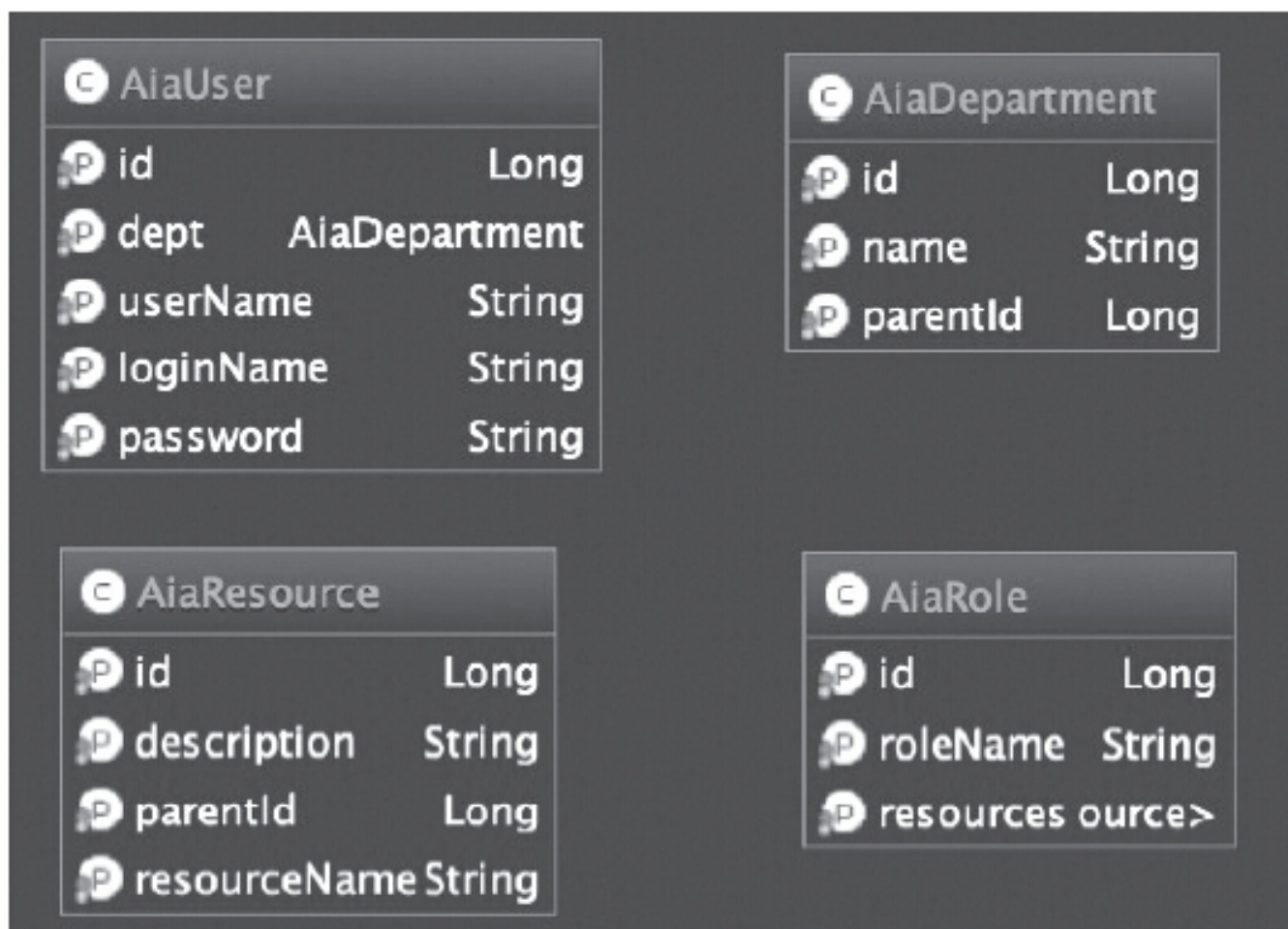


图19-2 与图19-1ER图对应的实体图

图19-3展示的是操控实体对象的接口，提供基本的CRUD功能，各个接口的说明如下：

- AiaUserManager: 用户实体管理接口。
- AiaDepartmentManager: 部门实体管理接口。
- AiaRoleManager: 角色实体管理接口，可以通过findByUserId()方法查询到用户拥有的角色列表。
- AiaResourceManager: 资源实体管理接口。

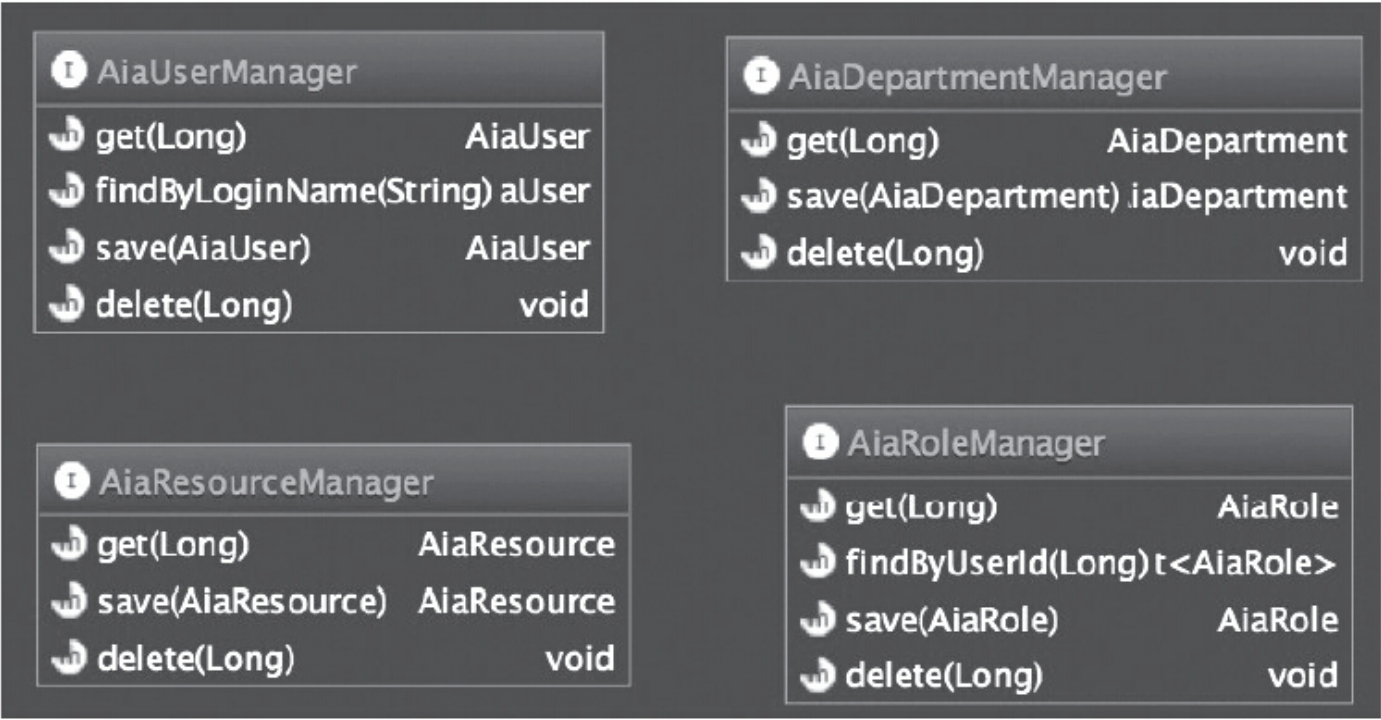


图19-3 操控图19-2中实体的接口定义

在5.1节中学习了如何使用IdentityService接口管理用户与组，以及用户与组的关系，假设现有系统已经提供了一套基于图19-1的数据结构的身份管理模块或独立系统，需要把这些数据同步到Activiti的身份数据表中或者通过配置为Activiti提供身份模块的数据，我们该如何实现呢？

笔者根据实施经验针对不同的情况提供了几种解决方案，读者可以据此选择合适的数据同步方式。

19.2 引擎身份接口方式

1.通过引擎接口同步数据

通过引擎接口同步数据是一种“非侵入式”的同步方式，做法类似于引擎中的监听，当现有身份模块的用户数据变更时调用Activiti引擎的IdentityService接口的对应方法同步操作，相关代码可以参考5.1节的内容。

当然在同步数据时要做好数据的校验工作，例如在添加、修改用户和组时要先检查对象是否存在防止引擎接口抛出异常。

下面针对图19-3中的接口定义实现具体的身份数据同步逻辑，代码清单19-1实现了AiaUserManager接口。以新增或更新用户数据为例，当管理员在身份管理系统中新增或更改一个用户时，通过Activiti的IdentityService接口把当前操作的用户对象同步到引擎的身份模块表中。其他几个接口就不再一一列举了，可以参考5.1节的内容实现AiaRoleManager接口同步角色及其与用户的关系。

代码清单19-1 AiaUserManager接口实现类

```
public class AiaUserManagerImpl implements AiaUserManager {
    private AiaUserDao dao;
    private IdentityService identityService;
    public AiaUser get(Long id) { return dao.get(id); }
    public AiaUser findByLoginName(String loginName) {
        return dao.findByLoginName(loginName);
    }
    public AiaUser save(AiaUser user) { // 新增或更新用户对象时同步数据到Activiti
        dao.save(user); // 持久化用户
        User activitiUser = null;
        if (user.getId() == null) { // 新增用户到Activiti身份模块
            activitiUser = identityService.newUser(user.getId().toString());
        } else { // 从引擎表查询已有用户
            activitiUser = identityService.createUserQuery()
                .userId(user.getId().toString()).singleResult();
            /** 省略代码 -> 复制user的属性到activitiUser */
            identityService.saveUser(activitiUser); // 保存用户
        }
        return user;
    }
    public void delete(Long id) { dao.delete(id); }
}
```

这种同步身份数据方式的特点在于“不破坏”引擎表结构、面向接口编程。如果根据用户ID查询相关的候选任务，那么SQL的过滤条件为：在ACT_RU_IDENTITYLINK表中类型为“candidate”且USER_ID_字段等于当前用户ID，或者任务有候选组且当前人与候选组有关联关系（在ACT_ID_MEMBERSHIP表中维护）。当我们操作业务系统的用户、角色以及两者的关系时把这些更改都同步到引擎的表中，所以引擎内部的SQL关联查询可以正常工作。之所以在这里强调用户与组的关系是因为下面介绍的第2种方式就不能使用与ACT_ID_MEMBERSHIP表关联查询了，因为下面的方式会禁用引擎的身份模块表（引擎不创建以ACT_ID开头的表）。

Activiti的每一张表都有一个对应的实体管理器，在引擎初始化时会初始化所有表的实体管理类（提供CRUD等功能），每一个实体类都有一个对应的实体管理类（XxxEntityManager）及实体管理工厂类（XxxEntityManagerFactory）。实体管理工厂类实现SessionFactory接口，以用户表的实体管理为例，用户实体类的实体管理工厂类如图19-4所示。图19-5则展示了用户实体管理类的继承图。

结合图19-4与图19-5可以了解到UserEntityManager类实现了Session与UserIdentity-Manager接口（定义了针对用户实体的CRUD方法），而UserEntityManagerFactory实现了SessionFactory接口并实现其两个方法，具体的实现代码见代码清单19-2所示。

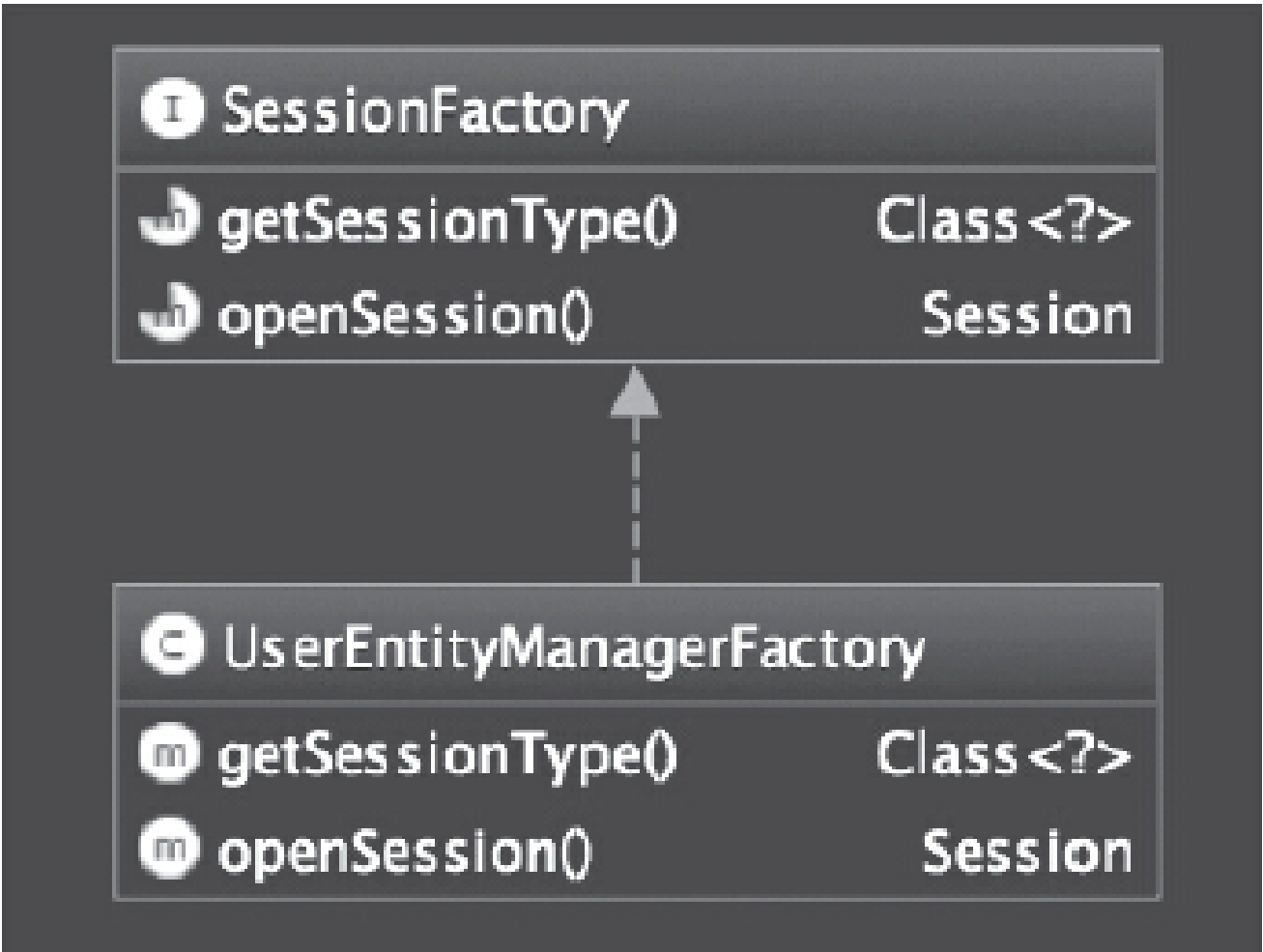


图19-4 用户实体管理工厂类User-EntityManagerFactory

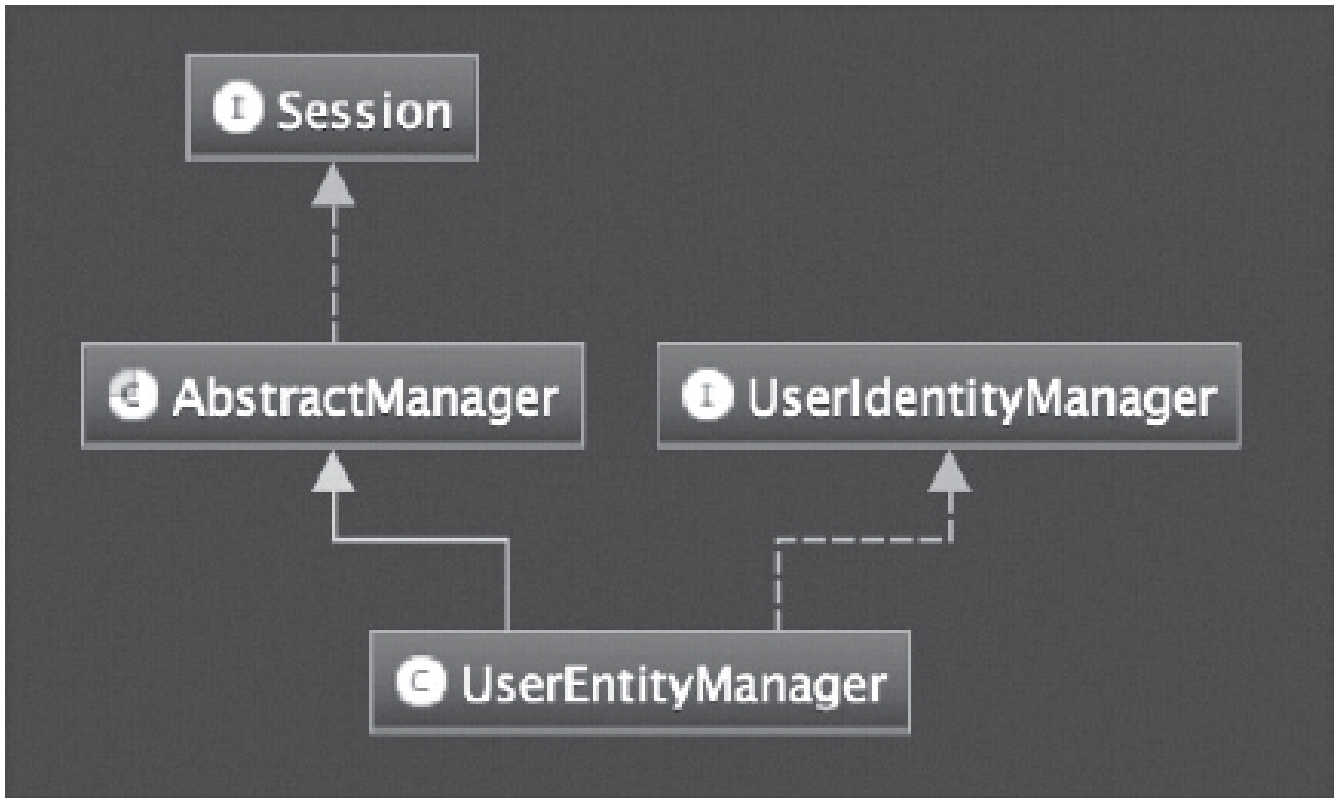


图19-5 用户实体管理类UserEntity-Manager继承图

代码清单19-2 UserEntityManagerFactory类

```

public class UserEntityManagerFactory implements SessionFactory {
    public Class< ? > getSessionType() {
        // 把UserIdentityManager接口注册到引擎
        // 由openSession()方法返回具体的实现类
        return UserIdentityManager.class;
    }
    // 创建一个新的实体管理类对象
    public Session openSession() { return new UserEntityManager(); }
}

```

实体管理的机制是为本节内容做的铺垫，引擎允许我们覆盖内置的实体管理器使用自定义的实体管理器替代默认实现类，以此我们可以替换与引擎和身份模块有关的实体管理类，把我们自定义的实体管理类注入引擎，自定义的实体管理类只要实现引擎的接口即可。下面的代码设置了引擎配置对象的customSessionFactory属性，并为该属性添加了两个实现了SessionFactory接口的bean对象。完整的配置参考配套资源中本章示例代码的src/resources/chapter19/applicationContext-session.xml。

```

<bean id="processEngineConfiguration" class="org.activiti.xx.XxxProcessEngineConfiguration">
    <!-- 组织机构适配 -->
    <property name="customSessionFactories">
        <list>
            <!-- 用户 (User) 实体管理器工厂 -->
            <bean class="me.kafeitu.activiti.chapter19.identity.session.AiaUserEntityManagerFactory">
                <property name="aiaUserEntityManager">
                    <bean class="me.kafeitu.activiti.chapter19.identity.session.AiaUserEntityManager">
                        <property name="aiaUserManager" ref="aiaUserManager"/>
                    </bean>
                </property>
            </bean>
            <!-- 组 (Group) 实体管理器工厂 -->
            <bean class="me.kafeitu.activiti.chapter19.identity.session.AiaGroupEntityManagerFactory">
                <property name="aiaGroupEntityManager">
                    <bean class="me.kafeitu.activiti.chapter19.identity.session.AiaGroupEntityManager">
                        <property name="aiaRoleManager" ref="aiaRoleManager"/>
                    </bean>
                </property>
            </bean>
        </list>
    </property>
</bean>
<!-- 用户实体管理器 -->
<bean id="aiaUserManager" class="me.kafeitu.activiti.chapter19.identity.impl.AiaUserManagerImpl">
    <property name="dao">
        <bean class="me.kafeitu.activiti.chapter19.identity.impl.AiaUserDaoImpl" />
    </property>
    <property name="identityService" ref="identityService"/>
</bean>
<!-- 角色实体管理器 -->
<bean id="aiaRoleManager" class="me.kafeitu.activiti.chapter19.identity.impl.AiaRoleManagerImpl">
    <property name="dao">
        <bean class="me.kafeitu.activiti.chapter19.identity.impl.AiaRoleDaoImpl"/>
    </property>
</bean>

```

上面的代码分别定义了两个实体管理工厂Bean并将它们加入到引擎的自定义Session工厂集合中，每一个工厂Bean均实现SessionFactory接口来声明该工厂的Session类型。代码清单19-3和代码清单19-4分别定义了用户 (User) 和组 (Group) 的实体管理Session工厂类。

代码清单19-3 AiaUserEntityManagerFactory类

```
public class AiaUserEntityManagerFactory implements SessionFactory {
    private AiaUserEntityManager aiaUserEntityManager;
    public void setAiaUserEntityManager(AiaUserEntityManager aiaUserEntityManager) {
        this.aiaUserEntityManager = aiaUserEntityManager;
    }
    public Class<?> getSessionType() { return UserIdentityManager.class; } #1
    public Session openSession() { return aiaUserEntityManager; } #2
}
```

代码清单19-4 AiaGroupEntityManagerFactory类

```
public class AiaGroupEntityManagerFactory implements SessionFactory {
    private AiaGroupEntityManager aiaGroupEntityManager;
    public void setAiaGroupEntityManager(AiaGroupEntityManager aiaGroupEntityManager) {
        this.aiaGroupEntityManager = aiaGroupEntityManager;
    }
    public Class<?> getSessionType() { return GroupIdentityManager.class; } #1
    public Session openSession() { return aiaGroupEntityManager; } #2
}
```

在代码清单19-3和代码清单19-4中，#1处分别返回了不同的Session类型，引擎在获取Session时会通过Session的类型从Session池（一个Map对象）中获取；#2处则分别返回具体的Session实现类对象。Session实现类的定义见代码清单19-5和代码清单19-6所示。

代码清单19-5 自定义的用户实体管理器AiaUserEntityManager

```
public class AiaUserEntityManager extends UserEntityManager {
    private AiaUserManager aiaUserManager;
    public Boolean checkPassword(String userId, String password) {
        AiaUser aiaUser = aiaUserManager.get(new Long(userId));
        return aiaUser.getPassword().equals(password);
    }
    public void setAiaUserManager(AiaUserManager aiaUserManager) {
        this.aiaUserManager = aiaUserManager;
    }
}
```

代码清单19-5中覆盖了引擎内部的UserEntityManager的checkPassword()方法，该方法用来校验用户名和密码是否匹配：在覆盖的方法中调用企业内部的身份模块接口查询到用户对象并校验用户输入的密码是否匹配，checkPassword()方法在用户登录时会被调用。

代码清单19-6 自定义的组实体管理器AiaGroupEntityManager

```
public class AiaGroupEntityManager extends GroupEntityManager {
    private AiaRoleManager aiaRoleManager;
    public void setAiaRoleManager(AiaRoleManager aiaRoleManager) {
        this.aiaRoleManager = aiaRoleManager;
    }
    public List<Group> findGroupsByUser(String userId) {
        List<AiaRole> roles = aiaRoleManager.findByUserId(new Long(userId));
        List<Group> groups = new ArrayList<Group>(roles.size());
        for (AiaRole aiaRole : roles) { // 循环转换AiaRole为GroupEntity对象
            GroupEntity group = new GroupEntity();
            group.setName(aiaRole.getRoleName()); // 角色中文名称
            group.setId(aiaRole.getRoleCode()); // 角色英文名称
            groups.add(group);
        }
        return groups;
    }
}
```

在代码清单19-6中覆盖了引擎的组实体管理器GroupEntityManager的findGroupBy-User()方法，该方法可以根据用户ID查询拥有的角色集合。当然除了这个方法还可以根据需要覆盖其他的方法，这里就不一一列举了。

19.3 用视图代替物理表

在引擎中，与身份相关的表如下：

- ACT_ID_USER：用户表。
- ACT_ID_INFO：用户信息表。
- ACT_ID_GROUP：组，也可以理解为角色。
- ACT_ID_MEMBERSHIP：用户与组的关系，这也是查询任务候选人需要关联的表。

相比前两种方式，使用视图代理物理表显得“轻量”一些，把引擎的物理表删除，取而代之的是与物理表同名的视图，只要保证视图的表结构与原来的物理表的表结构及字段类型保持一致即可。

在删除了引擎身份模块的物理表后启动会报错，提示缺少身份模块的表，异常信息如下：

```
org.activiti.engine.ActivitiException: Activiti database problem: Tables missing for component(s) identity
```

从表面看，是因为缺少身份模块的表。但是有同名的视图存在为什么引擎依然提示缺少表呢？这是因为引擎会检查“表”的类型，虽然名称相同，但是类型不是物理表（table）而是视图（view）。不过Activiti开发团队可能早就料到我们会抛弃这个简陋的身份模块，所以我们开启了一道大门，允许我们放弃身份模块也就是说在初始化引擎时不再检查ACT_ID_*表是否存在，相当于禁用了身份模块的功能。下面的代码把属性“dbIdentityUsed”设置为“false”，即禁用了身份模块的功能。

```
<bean id="processEngineConfiguration" class="org.activiti.spring.SpringProcessEngineConfiguration">
    ...
    <property name="dbIdentityUsed" value="false"/>
</bean>
```

19.4 集成LDAP

LDAP (Lightweight Directory Access Protocol, 轻量目录访问协议) 被用来管理地址簿或人员与资源的关系。一般当企业中有多个业务系统时会使用LDAP统一管理用户资源, 各个系统不需要单独维护而是通过LDAP协议获取所需用户数据。

Activiti从5.13版本开始提供了activiti-ldap模块, 用来整合Activiti与LDAP服务, 也就是放弃Activiti的身份模块, 当引擎需要读取用户、组数据时调用LDAP接口访问。但是activiti-ldap模块只提供用户数据查询功能, 不支持有关数据更改的操作 (新增、删除、更新等), 例如验证用户名和密码是否匹配以及查询用户有哪些候选 (Candidate) 任务。

在Activiti中集成LDAP服务也比较简单, 添加依赖的Jar包后在引擎配置对象中注入activiti-ldap模块的配置类[org.activiti.ldap.LDAPConfigurator](#)。本章的Spring配置文件中加入了针对LDAP服务的配置代码, 相关配置见代码清单19-7所示。

代码清单19-7 Activiti引擎配置对象中的LDAP配置

```
<!-- 嵌入LDAP服务器 (从classpath环境中读取) -->
<security:ldap-server ldif="classpath:chapter19/users.ldif" root="o=aia" /> #1
<bean id="processEngineConfiguration" // 可以配置所有实现了引擎配置接口的类
class="org.activiti.spring.SpringProcessEngineConfiguration">
http://www.hzcourse.com/resource/readBook?path=/openresources/teach_ebook/uncompressed/15030/OEBPS/Text/...
<property name="configurators"> // 把LDAP配置对象注入给引擎
<list>
<bean class="org.activiti.ldap.LDAPConfigurator"> // LDAP配置对象
<!--LDAP服务器配置参数 -->
<property name="server" value="ldap://localhost" /> // LDAP服务地址
<property name="port" value="33389" /> // LDAP服务端口
<property name="user" // 访问LDAP的账号
value="uid=admin, ou=users, o=aia" />
<property name="password" value="pass" /> // 访问LDAP的密码
<property name="initialContextFactory" // ContextFactory工厂类 (默认值)
value="com.sun.jndi.ldap.LdapCtxFactory" />
//连接LDAP时设置的"java.naming.security.authentication"属性值 (默认值)
<property name="securityAuthentication" value="simple" />
<property name="customConnectionParameters">
<map // 可以设置自定义连接参数, 例如连接池、安全属性等
<entry key="key" value="value"></entry>
</map> // 参见http://docs.oracle.com/javase/tutorial/jndi/ldap/jndi.html
</property>
<!--查询数据的过滤条件-->
// Distinguished?Name, 查询数据的基础名称, 相当于数据库中的主键
<property name="baseDn" value="o=aia" />
// 查询用户的DN, 如果不设置该属性, 默认使用baseDn属性的值
<property name="userBaseDn" value="ou=users,o=aia" />
// 查询组的DN, 如果不设置该属性, 默认使用baseDn属性的值
<property name="groupBaseDn" value="ou=groups,o=aia" />
<property name="queryUserById" // 根据用户ID查询语句
value="{&(objectClass=inetOrgPerson)(uid={0})}" />
<property name="queryUserByFullNameLike" // 全名查询语句
value="{&(objectClass=inetOrgPerson)(l({0}={1})*({2}={3}*))}" />
<property name="queryGroupsForUser" // 查询用户所属组语句
value="{&(objectClass=groupOfUniqueNames)(uniqueMember={0})}" />
<!--LDAP数据源中资源相关属性定义 -->
<property name="userIdAttribute" value="uid" /> // 匹配用户ID
<property name="userFirstNameAttribute" value="cn" /> // 匹配用户的姓
<property name="userLastNameAttribute" value="sn" /> // 匹配用户的名
<property name="groupIdAttribute" value="gid" /> // 匹配组ID的属性名
<property name="groupNameAttribute" value="cn" /> // 匹配组名称的属性名
</bean>
</list>
</property>
</bean>
```

代码清单19-7列出了有关LDAP的配置, 完整的配置可以参考本章示例目录的[src/main/resources/applicationContext.xml](#)文件。本章示例代码由#1处嵌入的LDAP服务器提供数据, 用户和组数据定义在本章源码的[src/main/resources/chapter19/users.ldif](#)文件中。

除了代码清单19-7中列出来的一些基本配置属性之外, activiti-ldap模块还支持高级配置, 这样可以由用户自定义内部的实现覆盖默认实现类, 例如如何查询用户、组。activiti-ldap支持覆盖的属性如表19-1所示。

表19-1 activiti-ldap模块的高级属性

属性名称	作用	默认值
IdapUserManager-Factory	代替查询用户工厂类 LDAPUserManager-Factory	org.activiti.ldap. LDAPUserManager-Factory
IdapGroupManager-Factory	代替查询组工厂类 LDAPGroupManager-Factory	org.activiti.ldap. LDAPGroupManager-Factory
IdapMembership-ManagerFactory	代替查询用户与组的工厂类 LDAP-MembershipManagerFactory	org.activiti.ldap. LDAPMembership-ManagerFactory
IdapQueryBuilder	自定义查询构造器, 可以替换查询 LDAP 数据的方法	org.activiti.ldap. LDAPQueryBuilder
groupCacheSize	组缓存的大小, 如果值小于 0, 就不会创建缓存	-1, 不缓存
groupCacheExpirationTime	设置组缓存的过期时间, 单位为毫秒。当获取特定用户的组时, 如果组缓存也启用了, 组会保存到缓存中, 并使用这个属性设置的时间。例如, 如果组在 00:00 被获取, 过期时间为 30 分钟, 那么所有在 00:30 之后进行的查询都不会使用缓存, 而是再次去 LDAP 查询。因此, 所以在 00:00-00:30 进行的查询都会使用缓存	1 小时

19.5 本章小结

本章讲解了几种已有用户数据与Activiti身份模块同步的方式，分为“侵入”式与“非侵入”式，具体介绍了通过引擎接口同步数据、自定义Session工厂，以及视图图代替物理表的方式。本章还介绍了如何从LDAP查询用户、组数据。读者可以根据不同的需求以及环境选择合适的同步方式。

第20章 REST服务

在企业应用中，常见的系统或平台都会提供多种接口方式，例如标准的Java API、TCP协议、基于SOAP（Simple Object Access Protocol，简单对象访问协议）的Web Service，以及REST（Representational State Transfer，表述性状态转移）软件风格等，一般这种软件风格被称为Restful。

作为一个BPM平台Activiti提供了基于Restful风格的API，从而可以通过HTTP协议访问Activiti的REST API实现流程的各种操作，同时也做到了跨平台、跨语言的通信。假如已有基于.NET或PHP平台开发的应用需要使用工作流，可以通过HTTP协议访问Activiti的REST API来操控流程引擎，使用方无须关心流程引擎使用哪种语言实现，这样就形成了一个流程中心，所有应用的流程都存储在一个数据库中，方便统一管理。

20.1 通信协议简介

Web Service是常用的跨平台数据交互技术之一，它的三个要素有：SOAP、WSDL（WebService Description Language）、UDID（Unique Device Identifier）。Web Service使用XML语言定义WSDL，用来描述Web Service中包含的方法及其参数和返回值信息。XML是一种通用语言，各个不同的厂商、语言只要遵循同一套标准，通过XML-RPC方式（Remote Procedure Call Protocol，远程过程调用协议）即可实现跨平台通信。

1999年6月W3C和IETF组织公布了RFC 2616，也就是现今使用最广泛的HTTP 1.1协议。在这个版本中定义了九种方法（也可以称之为“动作”）来区分URI指定的资源处理方式，包括OPTIONS、HEAD、GET、POST、PUT、DELETE、TRACE、CONNECT、PATCH。关于每一种方法的解释可以参考维基百科：<http://zh.wikipedia.org/zh/超文本传输协议>。

REST最早是由Roy Thomas Fielding博士在2000年的学术论文《Architectural Styles and the Design of Network-based Software Architectures》中提出的，它是一种针对网络应用设计的架构模式，从表面上看能让客户端请求的URI更加友好化，从系统开发层面来讲可以降低系统的复杂性，从而提升系统的可伸缩性。

一个典型的REST架构的URI大致如下（描述的是有关IT类的数目）。

<code>http://www.site.com/book/it</code>
当使用GET方法访问时会列出所有的书目；当使用POST方法访问时会新增一本书目；当使用DELETE方法访问时会删除所有IT类的数目。
下面的这个URI与上面的类似，但是作用不同。
<code>http://www.site.com/book/it/321</code>

这个URI中的321表示书的ID，当使用GET方法访问时可以得到与ID对应的书的介绍信息，例如书名、作者、ISBN、价格等，可以选择任意的网络媒体格式（例如XML、JSON）。当使用DELETE方法时会删除ID为321的图书。

有关REST协议的详细介绍可以参考维基百科：<http://zh.wikipedia.org/zh/REST>。

20.2 REST API概述

目前大家对基于Java语言的REST框架已经比较成熟，例如Apache CXF、Restlet等。Activiti 5.16.3版本之前使用了Restlet作为REST基础框架，读者可以在官方源码的基础上方便地通过二次开发进行功能扩展。从5.16.4版本开始由Restlet切换到了Spring MVC作为REST基础框架，但是在5.16.4版本中并没有修改REST API的资源URL，仅仅把实现框架做得更好而已。表20-1列出了REST API提供的几大类资源，每一个大类都用一个单词作为资源的基础URI。

表20-1 Activiti REST API 资源

资源分类	资源基础URI	说明
引擎管理（Deployments）	management	用来获取引擎信息、管理作业（Job）、查询引擎的数据库表等
流程仓库（Repository）	repository	用来部署、查询、删除流程定义，创建、读取模型（Model）等
运行时（Runtime）	runtime	用来管理运行时流程实例、任务、变量、事件、批注等

(续)

资源分类	资源基础 URI	说 明
历史（History）	history	用来管理历史流程实例、任务、变量、批注等
表单（Form）	form	用来读取表单内容、提交任务表单数据
身份（Identity）	identity	用来管理用户、组数据以及用户名和密码校验等功能
查询（Query）	query	查询运行时及历史流程实例、任务等

Restful风格的URI很直观易懂，按照资源的层级依次拼接，以不同的HTTP方法请求触发被请求资源的不同动作。关于完整的REST API定义可以查看用户手册中的REST部分，其中详细介绍了每一个资源的作用、接收的参数以及返回结果等。

下面展示了根据任务ID获取任务对象的URI。

```
http://localhost:8080/activiti-rest/runtime/tasks/{taskId}
```

这个URI中的activiti-rest是REST API的应用名称，runtime是运行时类资源的基础URI，tasks是任务类型的资源，{taskId}表示需要在使用时替换成实际的任务ID（与本书中的Spring MVC控制器的@RequestMapping注解语法类似）。下面的URI用数字125代替taskId，表示读取任务ID为125的任务对象。

```
http://localhost:8080/activiti-rest/runtime/tasks/125
```

请求该URI后“应该”（稍后解释）会得到类似以下的JSON数据：

```
{
  "id": "4396",
  "url": "http://localhost:8080/aia/rest/runtime/tasks/4396",
  "owner": null,
  "assignee": "kafeitu",
  "delegationState": null,
  "name": "调整申请",
  "description": null,
  "createTime": "2014-01-23T08:31:09.894+0000",
  "dueDate": null,
  "priority": 50,
  "suspended": false,
  "taskDefinitionKey": "modifyApply",
  "tenantId": "",
  "category": "",
  "parentTaskId": null,
  "parentTaskUrl": null,
  "executionId": "4370",
  "executionUrl": "http://localhost:8080/aia/rest/runtime/executions/4370",
  "processInstanceId": "4370",
  "processInstanceUrl": "http://localhost:8080/aia/rest/runtime/process-instances/4370",
  "processDefinitionId": "leave:1:3020",
  "processDefinitionUrl": "http://localhost:8080/aia/rest/repository/process-definitions/leave%3A1%3A3020",
  "variables": []
}
```

事实并不是我们所期望的那样，直接访问上面的URI会得到一个HTTP 401错误提示。熟悉HTTP响应代码的读者应该知道401表示认证失败，也就是用户没有权限访问所请求的URI。因此，在没有提供认证信息之前访问上面的URI得到的结果如下：

```
{
  "errorMessage": "Authentication is required",
  "statusCode": 401
}
```

REST一个很大的特性就是无状态，读者可以与常规系统进行对比来理解。一般的业务系统都会用户在登录成功后保存用户信息到HttpSession对象中，这样所有的请求需要使用用户信息时直接从Session对象中获取即可。REST则不然，其无状态特性把一个WEB请求与其他的请求完全隔离，资源请求者需要一次提供资源所需的全部信息，例如提供用户认证信息、请求参数等。在一般的REST架构服务中，都会把用户认证信息放在请求信息的HTTP头部（Header），在头部信息中添加一个Base Authorization认证，或者在URI中包含用户名和密码：

```
http://kermit:000000@localhost:8080/aia/rest/...
```

以上URI中kermit为用户名，冒号后面的000000为对应的密码。

20.3 发布REST API

Activiti的REST API在Restlet基础上进行开发，不了解Restlet的读者也不用担心。如果仅使用REST API，只需要部署Activiti官方提供的war包（需要解压缩）到Tomcat或其他容器中即可。

从官方下载的压缩包中（例如activiti-5.15.1.zip）包含一个名称为wars的目录，该目录包含两个war文件：activiti-explorer.war、activiti-rest.war。activiti-explorer的功能在前面已经介绍，而activiti-rest就是Activiti的REST API。

用压缩工具把activiti-rest.war文件解压复制到Tomcat的webapp目录中。如果要修改REST API访问的数据库，可以打开activiti-rest/WEB-INF/classes/db.properties文件修改配置，默认的数据库配置如下：

```
db=h2
jdbc.driver=org.h2.Driver
jdbc.url=jdbc:h2:mem:activiti;DB_CLOSE_DELAY=-1
jdbc.username=sa
jdbc.password=
```

 **说明** 与activiti-explorer一样，启动activiti-rest模块时会初始化用户数据。

虽然使用内存数据库没有实际意义，所以把数据库配置文件中的jdbc.url属性修改为第17章示例代码的jdbc配置的地址。

```
jdbc.url=jdbc:h2:file:~/activiti-in-action-chapter17
```

启动容器后就可以请求REST API了。访问方式有很多种，可以通过浏览器直接访问，也可以通过Restlet、Apache Cxf、Apache HttpClient访问。

在activiti-rest应用中配置了Restlet的Servlet来拦截请求，相关配置如下：

```
<!-- Restlet adapter -->
<servlet>
  <servlet-name>RestletServlet</servlet-name>
  <servlet-class>
    org.restlet.ext.servlet.ServerServlet
  </servlet-class>
  <init-param>
    <param-name>org.restlet.application</param-name>
    <param-value>
      org.activiti.rest.common.application
        .ActivitiRestApplication
    </param-value>
  </init-param>
</servlet>
<!-- Catch all requests -->
<servlet-mapping>
  <servlet-name>RestletServlet</servlet-name>
  <url-pattern>/service/*</url-pattern>
</servlet-mapping>
```

20.3.1 通过浏览器访问

在浏览器中输入如图20-1中的URL，查看所有的引擎属性，按回车后浏览器会弹出一个对话框提示输入用户名和密码验证身份，输入完用户名和密码后点击“Login”按钮，这时REST模块给出了响应，如图20-2所示。

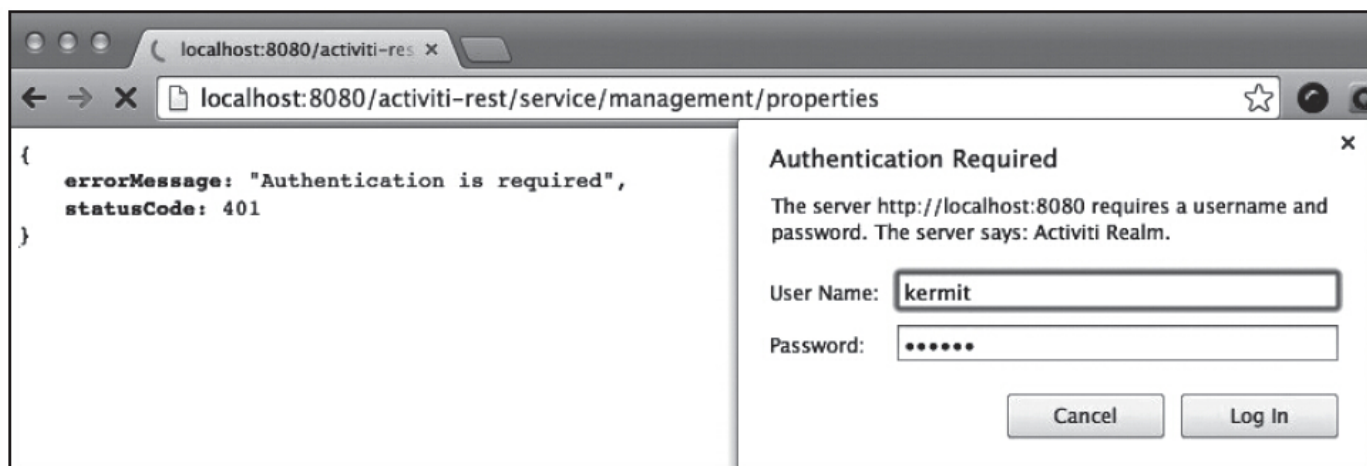


图20-1 从浏览器发送请求提示验证用户身份



图20-2 REST API的响应结果

从图20-2中可以看出，REST API给出了正确的响应结果，用JSON数据格式展示了引擎的三个属性，实际上是调用了ManagementService的getProperties()方法，然后由Restlet转换为JSON字符串输出给客户端。

也可以实验一下在HTTP协议后面直接传递用户名和密码验证身份，对应的URL如下：

```
http://kermit:kermit@localhost:8080/activiti-rest/service/management/properties
```

20.3.2 通过HttpClient访问

Apache HttpClient是常用的HTTP协议访问工具包，简单的几行代码就可以完成一次请求并接收响应结果。代码清单20-1列出了通过Apache HttpClient访问REST API的代码，用来读取引擎的配置属

性。

代码清单20-1 通过HttpClient访问REST API

```
public class RestRequestByHttpClient {
private static String REST_URI =
    "http://localhost:8080/activiti-rest/service/management/properties";
    public static void main(String[] args) throws IOException {
        // 设置Base Auth验证信息
        CredentialsProvider provider = new BasicCredentialsProvider();
        UsernamePasswordCredentials credentials =
            new UsernamePasswordCredentials("kermit", "kermit");
        provider.setCredentials(AuthScope.ANY, credentials);
        // 创建HttpClient
        HttpClient client = HttpClientBuilder.create()
            .setDefaultCredentialsProvider(provider).build();
        // 发送请求
        HttpResponse response = client.execute(new HttpGet(REST_URI));
        HttpEntity entity = response.getEntity(); // 接收响应实体
        if (entity != null) { // 编码转换后输出到控制台
            String content = EntityUtils.toString(entity, "UTF-8");
            System.out.println(content);
        }
    }
}
```

执行完代码清单20-1中的类后在控制台打印了REST API返回的结果：

```
{ "next.dbid": "101", "schema.history": "create(5.15)", "schema.version": "5.15" }
```

20.3.3 通过Restlet访问

Restlet项目是一个开源轻量级的REST框架实现，功能比较全面，所以在企业中也是常用的REST框架。代码清单20-2列出了通过Restlet访问REST API的代码。

代码清单20-2 通过Restlet访问REST API

```
public class RestRequestByRestlet {
    private static String REST_URI =
        "http://localhost:8080/activiti-rest/service/management/properties";
    public static void main(String[] args) throws IOException {
        // 创建Resource对象
        ClientResource resource = new ClientResource(REST_URI);
        // 设置Base Auth认证
        resource.setChallengeResponse(ChallengeScheme.HTTP_BASIC, "kermit", "kermit");
        Representation representation = resource.get();
        ObjectMapper mapper = new ObjectMapper();
        // 把文本转换为JSON格式对象
        JsonNode jsonNode = mapper.readTree(representation.getStream());
        // 循环输出响应内容
        IteratorString fieldNames = jsonNode.getFieldNames();
        while (fieldNames.hasNext()) {
            String fieldName = fieldNames.next();
            System.out.println(fieldName + " : " + jsonNode.get(fieldName));
        }
    }
}
```

执行完代码清单20-2中的类后输出结果如下：

```
next.dbid : "101"
schema.history : "create(5.15)"
schema.version : "5.15"
```

20.3.4 通过Apache CXF访问

Apache CXF是目前比较流行的一个开源Service框架。CXF可以使用WSDL标准定义并支持多种消息格式，例如SOAP、XML/HTTP等。CXF除了支持标准的SOAP RPC的Web Service之外，还可以发布REST架构的Web Service（Restful风格）。代码清单20-3列出了使用Apache CXF访问REST API的代码。

代码清单20-3 通过Apache CXF访问REST API

```
public class RestRequestByCxf {
private static String REST_URI =
    "http://localhost:8080/activiti-rest/service/management/properties";
    public static void main(String[] args) throws IOException {
        // 创建client对象
        WebClient client = WebClient.create(REST_URI);
        // Basic Auth身份认证
        String auth = "Basic " + Base64Utility.encode("kermit:kermit".getBytes());
        client.header("Authorization", auth);
        // 获取响应内容
        Response response = client.get();
        InputStream stream = (InputStream) response.getEntity();
        // 转换并输出响应结果
        StringWriter writer = new StringWriter();
        IOUtils.copy(stream, writer, "UTF-8");
        String respText = writer.toString();
        System.out.println(respText);
    }
}
```

代码清单20-3执行完成后的结果如下：

```
{ "next.dbid": "101", "schema.history": "create(5.15)", "schema.version": "5.15" }
```

20.4 集成REST API

假如某个应用以嵌入式集成了Activiti引擎，除了在互联网调用引擎接口外还需要对外提供接口服务，此时REST API就派上了用场，只需要简单配置即可对外提供标准的REST服务接口，同时也节省了开发成本。

读过源码的读者会发现源码目录中和REST有关的模块有以下三个：

- activiti-common-rest，公共对象以及工具类。
- activiti-rest，具体的API实现及定义。
- activiti-webapp-rest2，依赖以上两个模块，可以打包成war部署到容器中发布。

从官方下载的压缩文件中的activiti-rest.war就是由activiti-webapp-rest2模块打包的结果。

要把REST API集成到现有的应用中只需要把activiti-rest-5.x.jar和activiti-common-rest-x.jar复制到应用的lib目录或使用Maven配置依赖（参考chapter20-rest目录中的pom.xml中定义的依赖），然后在应用的web.xml中配置相关的Servlet映射即可发布REST接口。

可以从官方源码或解压的activiti-rest.war中复制以下两个文件到项目源码的资源根目录中，如图20-3所示。

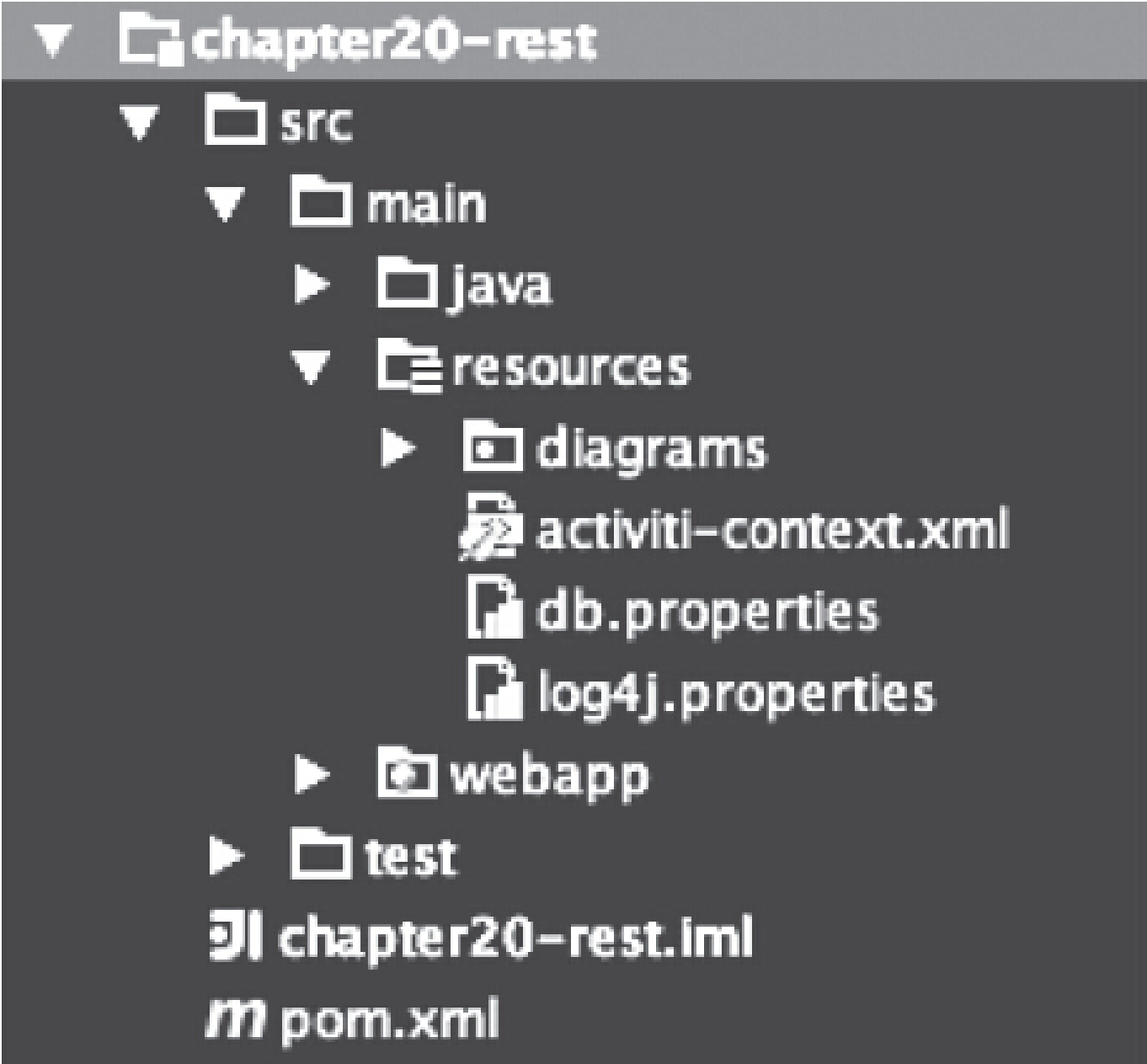


图20-3 包含了REST API所需的配置文件

- activiti-context.xml，引擎的配置文件（用Spring代理引擎）。
- db.properties，引擎数据库属性配置文件。

文件activiti-context.xml不需要修改，当然也可以添加自己的配置信息，要配置数据库可以修改db.properties文件。

在准备好了引擎的配置文件后，接下来要在应用的web.xml中加入REST API的相关配置，包括Activiti引擎上下文初始化监听器（ActivitiServletContextListener）以及Servlet映射，具体配置如下：

```
<listener>
  <listener-class> // 上下文监听器
    org.activiti.rest.common.servlet
      .ActivitiServletContextListener
  </listener-class>
</listener>
```

```
<!-- Restlet 适配器 -->
<servlet>
  <servlet-name>RestletServlet</servlet-name>
  <servlet-class>
    org.restlet.ext.servlet.ServerServlet
  </servlet-class>
  <init-param>
    <!-- 定义路由 -->
    <param-name>org.restlet.application</param-name>
    <param-value>org.activiti.rest.service.application
      .ActivitiRestServicesApplication
    </param-value>
  </init-param>
</servlet>
<!-- Catch all requests -->
<servlet-mapping>
  <servlet-name>RestletServlet</servlet-name>
  <url-pattern>/service/*</url-pattern> // 接口访问基路径
</servlet-mapping>
```

20.4.2 通过Ajax访问

在20.3节中介绍了可以使用Java代码访问REST接口，还可以使用前端技术Javascript来访问，这样只要有浏览器就可以向引擎发送请求，下面的代码列出了通过jQuery.ajax读取引擎属性。

```
$.ajax({
  type: "get",
  url: "http://localhost:8080/chapter20-rest/
    service/management/properties",
  dataType: "json",
  beforeSend: function(xhr) { // 设置认证信息
    var token = 'kermit:000000';
    var hash = Base64.encode(token);
    var baseAuth = "Basic " + hash;
    xhr.setRequestHeader('Authorization', baseAuth);
  },
  contentType: "application/json",
  success: function(json) {
    $.each(json, function(k, v) {
      $('<li>', {
        html: k + ": " + v
      }).appendTo('#resp');
    });
  }
});
```

在以上代码中，beforeSend定义的function用于设置认证信息，在通过Base64函数加密后设置到XmlHttpRequest对象的header中，请求的参数与返回的结果类型均为JSON类型。

在启动本章示例代码后，在浏览器中打开<http://localhost:8080/chapter20-rest>，单击图20-4中的“Ajax访问”按钮后显示了引擎的属性信息。

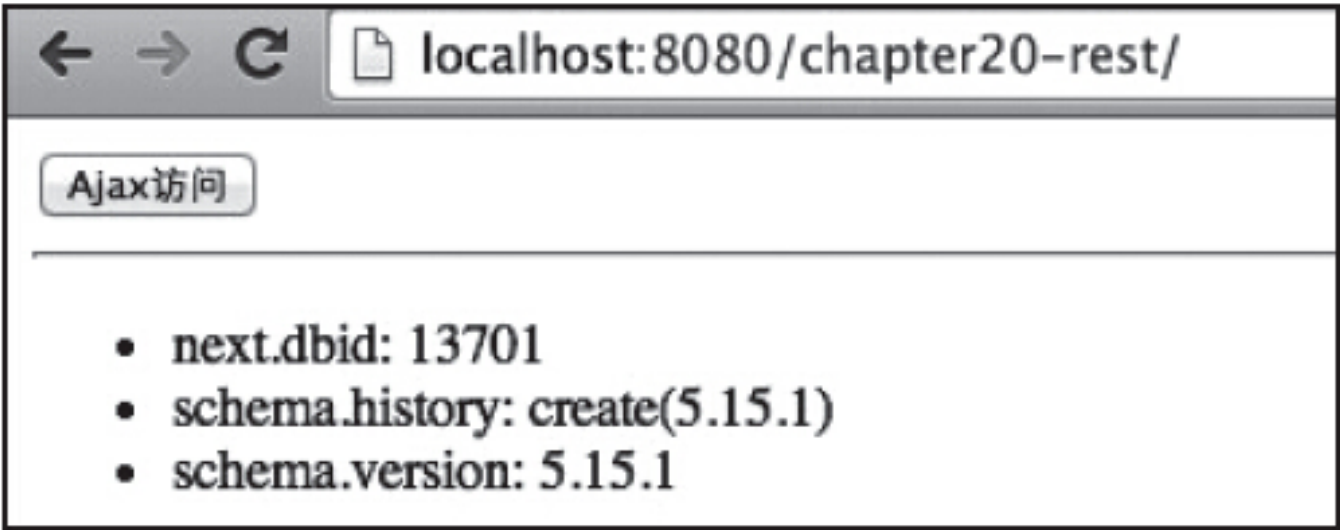


图20-4 通过Ajax访问REST API

要实现Ajax跨域访问，需要使用JSONP方式，不过由于JSONP不支持Basic Authorization认证，因此需要自行实现，例如在Restlet中配置支持跨域（Access-Control-Allow-Origin），或者替换原有的Basic Authorization身份认证方式。

20.5 完整示例

本节内容将使用REST API完成一个完整的流程示例，涉及几个典型的操作，例如部署流程、查询流程定义、任务查询、任务完成等。

本节的示例代码以Apache CXF作为访问REST API的客户端。由于示例涉及的内容比较多，因此会分段介绍，完整的示例代码可以查看本书配套资源中的文件chapter20-rest/src/main/java/me/kafeitu/activiti/chapter20/rest/RestRequestTandem.java。

在运行示例代码之前先启动Activiti官方提供的activiti-rest.war服务，示例代码中涉及REST_BASE_URI的地方均使用下面的常量定义：

```
private static final String BASE_REST_URI =
  "http://localhost:8089/activiti-rest/service/";
```

代码清单20-4列出了几个通用方法辅助示例，例如创建WebClient对象和结果输出，方便读者分析请求、响应的JSON数据。

代码清单20-4 完整示例中几个公共方法

```
// 创建Client
private static WebClient createClient(String uri) {
    WebClient client = WebClient.create(BASE_REST_URI + uri);
    String auth = "Basic " + Base64Utility.encode("kermit:kermit".getBytes());
    client.header("Authorization", auth);
    return client;
}
// 打印发送请求的JSON数据
private static void printJsonString(String phase, String json) {
    System.out.println("\n+++ 发送请求[" + phase + "] +++");
    System.out.println(json);
}
// 打印输出结果
private static JsonNode printResult(String phase, Response response) {
    System.out.println("\n=== " + phase + " ===");
    try {
        InputStream stream = response.getEntity();
        int available = stream.available();
        if (available == 0) {
            System.out.println("nothing returned, response code: " + response.getStatus());
            return null;
        }
        JsonNode responseNode = objectMapper.readTree(stream);
        if (formatOutputJson) {
            System.out.println(objectMapper.writerWithDefaultPrettyPrinter()
                .writeValueAsString(responseNode));
        } else {
            System.out.println(objectMapper.writeValueAsString(responseNode));
        }
        return responseNode;
    } catch (IOException e) {
        System.err.println("catch an exception: " + e.getMessage());
        e.printStackTrace();
    }
    return null;
}
```

20.5.1 部署流程

部署流程对应的资源路径为“repository/deployments”，支持GET、POST请求类型，代码清单20-5列出了部署流程的示例代码。

代码清单20-5 部署流程（创建Deployment对象）

```
private static void deployLeave() {
    WebClient client = createClient("repository/deployments"); // 创建WebClient对象
    InputStream resourceAsStream = RestRequestTandem.class.getClassLoader().getResourceAsStream("diagrams/leave.bpmn"); // 获取bpmn文件资源 #1
    client.type("multipart/form-data"); // 设置请求内容类型 #2
    ContentDisposition cd = new ContentDisposition("form-data;name=bpmn;filename=leave.bpmn"); #3-1
    Attachment att = new Attachment("leave.bpmn", resourceAsStream, cd); #3-2
    Response response = client.post(new MultiPartBody(att)); // 以POST方式发送请求 #4
    printResult("部署流程", response); // 转换并输出响应结果 #5
}
```

代码清单20-5的作用是上传文件流到REST服务，这与20.3.4节使用的API有一些差别：#1处从classpath中读取BPMN文件流；#2处设置请求的内容类型，由于本例涉及文件上传，因此必须声明请求的header属性“Content-Type”为“multipart/form-data”；#3-1处定义了内容的类型（name属性必须定义，否则REST API会抛出NullPointerException异常），接着在#3-2处定义了附件对象；#4处把附件作为请求的Body以POST请求发送给REST服务；#5处把相应结果打印在了控制台，结果如下：

```
=== 部署流程 ===
{
  "id" : "1004",
  "name" : "leave.bpmn",
  "deploymentTime" : "2014-06-14T07:33:50.329+0000",
  "category" : null,
  "url" : "http://localhost:8089/activiti-rest/service/repository/deploy-ments/1004",
  "tenantId" : ""
}
```

结果是一个标准的JSON格式数据，包含了Deployment的几个主要属性，其中“url”的值是获取的单个Deployment对象详细的URL。

20.5.2 查询Deployment

查询Deployment与部署流程的资源路径相同，只不过查询Deployment使用的是POST类型的请求，可以附加查询条件，例如名称（name）、分类（category）等。代码清单20-6列出了部署流程的示例代码。

代码清单20-6 查询Deployment

```
private static void queryDeployment() {
    WebClient client = createClient("repository/deployments");
    Response response = client.get(); // GET请求类型
    // 转换并输出响应结果
    printResult("查询Deployment", response);
}
```

代码清单20-6的输出结果如下：

```
{
  "data" : [ {
    "id" : "1004",
    "name" : "leave.bpmn",
    "deploymentTime" : "2014-06-14T07:33:50.329+0000",
    "category" : null,
    "url" : "http://localhost:8089/activiti-rest/service/repository/deploy-ments/1004",
    "tenantId" : ""
  } ],
  "total" : 1,
  "start" : 0,
  "sort" : "id",
  "order" : "asc",
  "size" : 1
}
```

代码清单20-6的输出结果与代码清单20-5相比多出了一些和分页有关的属性，这说明可以分页查询数据。通用的分页查询参数见下面的列表：

- sort: 排序字段名称。
- order: 排序类型，升序（asc）、降序（desc）。

- start: 起始记录值，默认值为0。
- size: 每一页显示的数量，默认值为10。

20.5.3 查询流程定义

查询流程定义与查询Deployment类似，只不过资源路径不同而已。代码清单20-7列出了查询流程定义的示例代码。

代码清单20-7 查询流程定义

```
private static JsonNode queryProcessDefinitions() {
    WebClient client = createClient("repository/process-definitions");
    Response response = client.get();
    // 转换并输出响应结果
    return printResult("查询流程定义", response);
}
```

代码清单20-7的输出结果如下:

```
=== 查询流程定义 ===
{
  "data" : [ {
    "id" : "leave:1:1007",
    "url" : "http://localhost:8089/activiti-rest/service/repository/process-definitions/leave%3A1%3A1007",
    "key" : "leave",
    "version" : 1,
    "name" : "请假流程-动态表单",
    "description" : "请假流程演示-动态表单",
    "deploymentId" : "1004",
    "deploymentUrl" : "http://localhost:8089/activiti-rest/service/repository/deployments/1004",
    "resource" : "http://localhost:8089/activiti-rest/service/repository/deployments/1004/resources/leave.bpmn",
    "diagramResource" : "http://localhost:8089/activiti-rest/service/repository/deployments/1004/resources/leave.leave.png",
    "category" : "http://www.kafeitu.me/activiti/leave",
    "graphicalNotationDefined" : true,
    "suspended" : false,
    "startFormDefined" : false
  } ],
  "total" : 1,
  "start" : 0,
  "sort" : "name",
  "order" : "asc",
  "size" : 1
}
```

代码清单20-7执行后输出的数据包含了分页信息，“data”属性为流程定义对象的数据集合，每一个流程定义都包含了必要的几个属性以及流程定义的XML、图片路径。

在浏览器中访问“diagramResource”属性值后（会提示输入用户名、密码）返回的JSON数据如图20-5所示。



图20-5 流程定义的图片资源信息

再次访问图20-5中所示“contentUrl”属性值后即可获取图片文件，如图20-6所示。

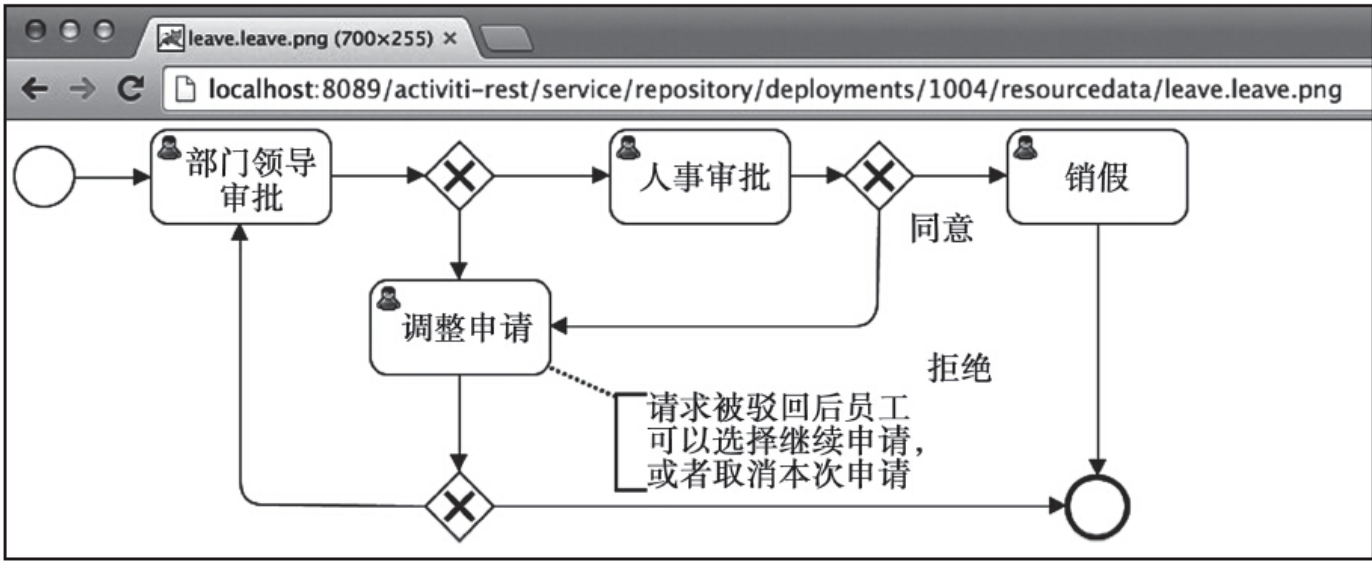


图20-6 通过REST服务获取的流程图

20.5.4 启动流程

启动流程的资源路径为“runtime/process-instances”，支持GET、POST类型的请求。GET请求可以查询流程列表（支持分页），POST请求可以启动一个流程实例。

启动流程的方式有多种：通过流程定义ID、通过流程KEY、通过触发消息（Message Start Event）。本节以流程定义ID的方式启动流程演示请求REST服务的过程。要通过流程定义ID启动流程，首先要得到流程定义ID。假设每次启动的都是用最新版本的流程，代码清单20-8列出了根据流程KEY读取最新版本流程定义ID的代码。

代码清单20-8 根据读取流程定义ID

```
public static void main(String[] args) throws IOException {
    // 查询到最新版本的请假流程，然后从结果集中获取data数据
    JsonNode jsonNode = queryLastVersionOfLeaveProcess();
    ArrayNode data = (ArrayNode) jsonNode.get("data");
    JsonNode next = data.iterator().next();
    String latestVersionId = next.get("id").asText();
    System.out.println("latestVersionId=" + latestVersionId);
}

private static JsonNode queryLastVersionOfLeaveProcess() {
    WebClient client = createClient("repository/process-definitions?key=leave&latest=true");
    Response response = client.get();
    // 转换并输出响应结果
    return printResult("查询最新版本的请假流程定义", response);
}
```

执行代码清单20-8后输出如下结果，从中可以看出最新版本的流程定义ID为“leave:1:1007”。

```
// 省略了JSON输出，和代码清单20-7的输出结果一样
latestVersionId=leave:1:1007
```

20.5.1节部署的请假流程是一个动态表单类型，所以需要在请求启动流程服务时提供启动表单字段值，对应的资源路径为“form/form-data”，可以用来获取指定流程定义或者任务的表单字段配置。代码清单20-9列出了获取指定流程定义的启动表单。

代码清单20-9 获取指定流程定义的启动表单

```
// 参数processDefinitionId由代码清单20-8中的latestVersionId变量提供
private static void queryProcessStartForm(String processDefinitionId) {
    String url = "Form/form-data?processDefinitionId=" + processDefinitionId;
    WebClient client = createClient(url);
    Response response = client.get();
    printResult("获取流程的启动表单属性", response);
}
```

执行代码清单20-9的结果如下，其中“formProperties”数组为启动流程的表单字段定义。

```
=== 获取流程的启动表单属性 ===
{
  "formKey" : null,
  "deploymentId" : "1004",
  "processDefinitionId" : "leave:1:1007",
  "processDefinitionUrl" : "http://localhost:8089/activiti-rest/service/repository/process-definitions/leave%3A1%3A1007",
  "taskId" : null,
  "taskUrl" : null,
  "formProperties" : [ {
    "id" : "startDate",
    "name" : "请假开始日期",
    "type" : "date",
    "value" : null,
    "readable" : true,
    "writable" : true,
    "required" : true,
    "datePattern" : "yyyy-MM-dd",
    "enumValues" : [ ]
  }, {
    "id" : "endDate",
    "name" : "请假结束日期",
    "type" : "date",
    "value" : null,
    "readable" : true,
    "writable" : true,
    "required" : true,
    "datePattern" : "yyyy-MM-dd",
    "enumValues" : [ ]
  }, {
    "id" : "reason",
    "name" : "请假原因",
    "type" : "string",
    "value" : null,
    "readable" : true,
    "writable" : true,
    "required" : true,
    "datePattern" : null,
    "enumValues" : [ ]
  } ]
}
```

现在可以根据获取的表单字段发送启动流程请求了（资源路径“form/form-data”），在内部调用FormService#getRenderedStartForm。代码清单20-10列出了通过REST的表单服务模块启动流程的示例代码。

代码清单20-10 启动流程——动态表单

```
private static JsonNode startProcessInstance(String processDefinitionId) throws IOException {
    WebClient client = createClient("form/form-data");
    client.type("application/json;charset=UTF-8"); // 内容类型必须设置
    Map<String, Object> parameters = new HashMap<String, Object>(); // #1-S
    // parameters.put("processDefinitionKey", "leave"); // #1-l
    parameters.put("processDefinitionId", processDefinitionId);
    List<Map<String, String>> variables = new ArrayList<Map<String, String>>();
    Map<String, String> var = new HashMap<String, String>();
    var.put("id", "startDate"); // 请假开始日期
    var.put("value", "2014-02-03");
    variables.add(var);
    var = new HashMap<String, String>();
    var.put("id", "endDate"); // 请假结束日期
    var.put("value", "2014-02-05");
    variables.add(var);
    var = new HashMap<String, String>();
    var.put("id", "reason"); // 请假原因
    var.put("value", "旅游");
    variables.add(var);
    parameters.put("properties", variables); // 设置表单属性 // #1-E
    String body = objectMapper.writerWithDefaultPrettyPrinter().writeValueAsString(parameters); // #2-S

    printJsonString("启动请假流程", body); // #2-E
    Response response = client.post(body);
    return printResult("启动请假流程", response);
}
```


#1处分别用Map对象设置了三个表单字段的值，最后在#1-E处把Map对象集合作为属性“properties”的值；#2-S处利用jackson工具把一个Map对象转换成格式化的JSON字符，参见下面执行结果中的“发送请求[启动请假流程]”部分；最后把JSON字符串作为请求内容通过POST请求发送给REST服务，返回了启动成功后的流程实例JSON数据。代码清单20-10的执行结果如下：

```
+++ 发送请求[启动请假流程] +++
{
  "properties" : [ {
    "id" : "startDate",
    "value" : "2014-02-03"
  }, {
    "id" : "endDate",
    "value" : "2014-02-05"
  }, {
    "id" : "reason",
    "value" : "旅游"
  } ],
  "processDefinitionId" : "leave:1:1007"
}
=== 启动请假流程 ===
{
  "id" : "1008",
  "url" : "http://localhost:8089/activiti-rest/service/runtime/process-instances/1008",
  "businessKey" : null,
  "suspended" : false,
  "processDefinitionId" : "leave:1:1007",
  "processDefinitionUrl" : "http://localhost:8089/activiti-rest/service/repository/process-definitions/leave%3A1%3A1007",
  "activityId" : "deptLeaderAudit",
  "variables" : [ ] ,
  "tenantId" : ""
}
```

上面的例子只是通过表单启动流程的一种方式，如果流程没有使用动态表单或不需要传递表单字段值，可以使用普通的流程启动方式。在向路径为“runtime/process-instances”的资源发送POST请求时就表示要启动一个流程，可以使用不同的方式，例如：流程定义KEY（processDefinitionKey）、流程定义ID（processDefinitionId）、消息名称（message）。详细的参数可以参考用户手册中“Start a process instance”部分的介绍。

20.5.5 读取流程变量

读取流程变量的资源路径为“runtime/process-instances/{processInstanceId}/variables”，在使用时把“{processInstanceId}”更换成具体的流程实例ID即可。代码清单20-11列出了读取流程变量的示例代码。

代码清单20-11 读取流程变量

```
private static void listProcessInstanceVariables(String processInstanceId) {
    String url = "runtime/process-instances/" + processInstanceId + "/variables";
    WebClient client = createClient(url);
    Response response = client.get();
    printResult("读取流程实例[" + processInstanceId + "]的变量", response);
}
```

代码清单20-11的执行结果如下。结果集中每一个变量分别用name、type、value、scope描述了变量的属性。

```
=== 读取流程实例[1008]的变量 ===
[ {
  "name" : "startDate",
  "type" : "date",
  "value" : "2014-02-02T16:00:00Z",
  "scope" : "local"
}, {
  "name" : "applyUserId",
  "type" : "string",
  "value" : "kermi",
  "scope" : "local"
}, {
  "name" : "reason",
  "type" : "string",
  "value" : "旅游",
  "scope" : "local"
}, {
  "name" : "endDate",
  "type" : "date",
  "value" : "2014-02-04T16:00:00Z",
  "scope" : "local"
} ]
```

20.5.6 查询任务

任务根据签收状态可以分为两种：未签收任务、已签收任务。这两种任务的REST服务的资源路径均为“runtime/tasks”，但使用的参数不同而已。关于具体的查询过滤参数，读者可以参考用户手册中的“List of tasks”部分。

在请假流程中，“部门领导审批”和“人事审批”均分配给不同的组（group），所以在过滤时可以选择使用候选人ID或候选组ID属性。对于“销假”节点，因为不需要签收而直接分配给流程发起人，所以要根据办理人（assignee）进行过滤。下面分别针对这两种类型进行介绍。

代码清单20-12列出了根据候选组查询任务的示例代码，使用的过滤参数名称为“candidateGroup”。

代码清单20-12 根据候选组查询任务

```
public static void main(String[] args) {
    JsonNode jsonNode = queryCandidateTask("deptLeader");
    data = (ArrayNode) jsonNode.get("data"); // 以下的代码用来获取任务的ID和名称
    next = data.iterator().next();
    String deptTaskId = next.get("id").asText();
    String taskName = next.get("name").asText();
    assert taskName.equals("部门领导审批");
}
private static JsonNode queryCandidateTask(String groupId) {
    WebClient client = createClient("runtime/tasks?candidateGroup=" + groupId);
    Response response = client.get();
    return printResult("读取组[" + groupId + "]的任务", response);
}
```

代码清单20-12的输出结果如下：

```
=== 读取组[deptLeader]的任务 ===
{
  "data" : [ {
    "id" : "1019",
```

```
    "url" : "http://localhost:8089/activiti-rest/service/runtime/tasks/1019",
    "owner" : null,
    "assignee" : null,
    "delegationState" : null,
    "name" : "部门领导审批",
    "description" : null,
    "createTime" : "2014-06-14T07:33:50.497+0000",
    "dueDate" : null,
    "priority" : 50,
    "suspended" : false,
    "taskDefinitionKey" : "deptLeaderAudit",
    "tenantId" : "",
    "category" : null,
    "parentTaskId" : null,
    "parentTaskUrl" : null,
    "executionId" : "1008",
    "executionUrl" : "http://localhost:8089/activiti-rest/service/runtime/executions/1008",
    "processInstanceId" : "1008",
    "processInstanceUrl" : "http://localhost:8089/activiti-rest/service/runtime/process-instances/1008",
    "processDefinitionId" : "leave:1:1007",
    "processDefinitionUrl" : "http://localhost:8089/activiti-rest/service/repository/process-definitions/leave%3A1%3A1007",
    "variables" : [ ]
  },
  "total" : 1,
  "start" : 0,
  "sort" : "id",
  "order" : "asc",
  "size" : 1
}
```

代码清单20-13中的方法用来查询指定用户的待办任务，与代码清单20-12不同，其过滤参数由 “candidateGroup” 换成了 “assignee” 。

代码清单20-13 根据办理人查询任务

```
public static void main(String[] args) {
    queryAssignedTask ("kermit");
}
private static JsonNode queryAssignedTask(String userId) {
    WebClient client = createClient("runtime/tasks?assignee=" + userId);
    Response response = client.get();
    return printResult("读取用户[" + userId + "]的任务", response);
}
```

代码清单20-13的输出结果与代码清单20-12的输出结果类似，读者可以自行运行示例查看。

20.5.7 签收任务

20.5.6小节介绍了如何查询候选任务，本小节接着介绍如何签收候选任务。在REST API中，签收 (claim)、提交 (complete)、委派 (delegate) 等操作都归属于任务动作 (task action)，资源路径为 “runtime/tasks/{taskId}”。有关任务动作的详细参数可以参考手册的 “Task actions” 部分，值得注意的是这些请求均没有响应内容，而是根据是否成功有不同的HTTP状态码，在使用时根据响应的HTTP状态码判断请求的结果。

代码清单20-14列出了签收任务的示例代码。

代码清单20-14 签收任务

```
public static void main(String[] args) {
    claimTask(taskId, "kermit");
}
private static void claimTask(String taskId, String userId) throws IOException {
    WebClient client = createClient("runtime/tasks/" + taskId);
    client.type("application/json;charset=UTF-8");
    Map<String, Object> parameters = new HashMap<String, Object>();
    parameters.put("action", "claim"); // 设置动作类型为claim
    parameters.put("assignee", userId); // 设置签收动作的人员ID
    String body = objectMapper.writerWithDefaultPrettyPrinter()
        .writeValueAsString(parameters);
    printJsonString("签收任务[" + taskId + "]", body);
    Response response = client.post(body);
    printResult("签收任务[" + taskId + "]", response);
}
```

代码清单20-14的输出结果如下，注意 “签收任务[1025]” 部分的输出结果，响应的状态码为200，这说明任务签收动作执行成功。

```
+++ 发送请求[签收任务[1025]] +++
{
  "action" : "claim",
  "assignee" : "lily"
}
== 签收任务[1025] ==
nothing returned, response code: 200
```

20.5.8 完成任务

完成任务与签收任务类似，资源路径同样为 “runtime/tasks/{taskId}”，不同的是action参数为 “complete”。代码清单20-15列出了完成任务的示例代码。

代码清单20-15 完成任务

```
public static void main(String[] args) {
    List<Map<String, String>> variables = new ArrayList<Map<String, String>>();
    Map<String, String> var = new HashMap<String, String>(); // 变量
    var.put("name", "deptLeaderApproved");
    var.put("value", "true");
    var.put("type", "string");
    variables.add(var);
    // 变量deptTaskId的值来源于代码清单20-12
    completeTask(deptTaskId, taskName, variables);
}
private static void completeTask(String taskId, String taskName,
    List<Map<String, String>> variables) throws IOException {
    WebClient client = createClient("runtime/tasks/" + taskId);
    client.type("application/json;charset=UTF-8");
    Map<String, Object> parameters = new HashMap<String, Object>();
    parameters.put("action","complete"); // 任务动作类型为complete
    if (variables != null) {
        parameters.put("variables", variables); // 设置变量
    }
    String body = objectMapper.writerWithDefaultPrettyPrinter()
        .writeValueAsString(parameters);
    printJsonString("完成任务[" + taskId + "-" + taskName + "]", body);
    Response response = client.post(body); // POST类型请求
    printResult("完成任务[" + taskId + "-" + taskName + "]", response);
}
```

代码清单20-15的执行结果如下，同样也是没有内容返回，是否成功需要根据HTTP响应码来判断。

```
+++ 发送请求 [完成任务 [1019-部门领导审批]] +++
{
  "taskId" : "1019",
  "properties" : [ {
    "name" : "deptLeaderApproved",
    "value" : "true"
  } ]
}
=== 完成任务 [1019-部门领导审批] ===
nothing returned, response code: 200
```

REST API中涉及的变量定义使用统一如下格式：

```
{
  "name" : "variableName",
  "value" : "variableValue",
  "valueUrl" : "http://http://www.hzcourse.com/resource/readBook?path=/openresources/teach_ebook/uncompressed/15030/OEBPS/Text/...",
  "scope" : "local",
  "type" : "string"
}
```

20.5.9 查询历史数据

前面几个示例都是与运行中（runtime）有关的，本小节简单介绍一下与历史（history）有关的示例。以查询历史流程实例为例，见代码清单20-16。

代码清单20-16 完成任务

```
private static JsonNode queryHistoricProcessInstance() {
    String url = "history/historic-process-instances?includeProcessVariables=true";
    WebClient client = createClient(url);
    Response response = client.get();
    return printResult("查询历史流程实例", response);
}
```

代码清单20-16的输出结果如下，url中的“includeProcessVariables=true”表示查询历史流程实例时附带相关的流程实例变量。有关历史对象查询的参数介绍可参看用户手册的“History”部分。

```
=== 查询历史流程实例 ===
{
  "data" : [ {
    "id" : "1008",
    "url" : "http://localhost:8089/activiti-rest/service/history/historic-process-instances/1008",
    "businessKey" : null,
    "processDefinitionId" : "leave:1:1007",
    "processDefinitionUrl" : "http://localhost:8089/activiti-rest/service/repository/process-definitions/leave%3A1%3A1007",
    "startTime" : "2014-06-14T07:33:50.496+0000",
    "endTime" : "2014-06-14T07:33:50.648+0000",
    "durationInMillis" : 152,
    "startUserId" : "kermit",
    "startActivityId" : "startevent1",
    "endActivityId" : "endevent1",
    "deleteReason" : null,
    "superProcessInstanceId" : null,
    "variables" : [
      ... // 省略了变量部分的输出
    ],
    "total" : 1,
    "start" : 0,
    "sort" : "processInstanceId",
    "order" : "asc",
    "size" : 1
  } ]
}
```

20.6 集成流程图跟踪组件Diagram Viewer

通过13.3节介绍的图片方式流程跟踪，可以直观地查看当前流程所处的节点（使用Javascript及CSS技术创建红色边框标示出来），这种方式依赖Activiti Designer自动生成的流程图。如果部署的流程没有提供对应的流程图（例如使用Activiti Modeler或自己开发的流程设计器），为了实现图片式流程跟踪可以使用Activiti官方提供的Diagram Viewer组件，该组件基于Javascript组件库Raphaël（一个使用SVG、VML的2D画图组件，兼容所有浏览器，其官网为：<http://dmitrybaranovskiy.github.io/raphael>）实现。

20.6.1 准备资源文件

Diagram Viewer组件被集成在Activiti Explorer中，解压activiti-explorer.war后的diagram-viewer目录结构如图20-7所示。

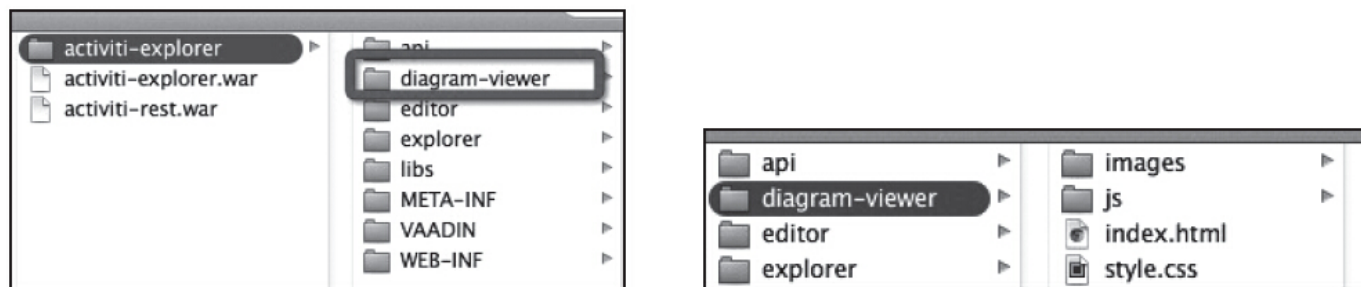


图20-7 activiti-explorer中集成的Diagram Viewer组件的目录结构

将图20-7中的diagram-viewer目录复制到chapter20-rest/src/main/webapp目录中。

20.6.2 准备配置文件

Diagram Viewer组件采用异步的方式 (Ajax) 请求REST模块提供的流程数据来展示流程图。这种方式需要REST模块提供流程中的各个组件以及坐标数据、流程实例的当前节点, 因此我们要为Diagram Viewer组件配置REST组件的服务地址。

在pom.xml文件中添加activiti-diagram-rest模块依赖, 依赖配置如下:

```
<dependency>
  <groupId>org.activiti</groupId>
  <artifactId>activiti-diagram-rest</artifactId>
  <version>${activiti-version}</version>
</dependency>
```

在web.xml (代码如下) 中配置继承自Restlet的Application类TraceRestApplication, 该类的实现可以在Activiti源码的activiti-common-rest模块中找到 (org.activiti.rest.explorer.application.ExplorerRestApplication) 。

```
<servlet>
  <servlet-name>TraceRestletServlet</servlet-name>
  <servlet-class>
    org.restlet.ext.servlet.ServerServlet
  </servlet-class>
  <init-param>
    <!-- Application class name -->
    <param-name>org.restlet.application</param-name>
    <param-value>
      me.kafeitu.activiti.chapter20.rest
      .application.TraceRestApplication
    </param-value>
  </init-param>
</servlet>
<servlet-mapping>
  <servlet-name>TraceRestletServlet</servlet-name>
  <url-pattern>/trace/*</url-pattern>
</servlet-mapping>
```

我们把这部分源码复制过来并将其改名为TraceRestApplication, 见代码清单20-17所示。

代码清单20-17 Diagram Viewer组件的服务入口

```
package me.kafeitu.activiti.chapter20.rest.application;
public class TraceRestApplication extends ActivitiRestApplication {
  public TraceRestApplication() {
    super();
  }
  @Override
  public synchronized Restlet createInboundRoot() {
    Router router = new Router(getContext());
    DiagramServicesInit.attachResources(router); // 添加创建流程图REST服务
    // 用jsonp包装以支持JSONP调用
    JsonpFilter jsonpFilter = new JsonpFilter(getContext());
    jsonpFilter.setNext(router);
    return jsonpFilter;
  }
}
```

20.6.3 访问Diagram Viewer跟踪流程

为了配合演示如何使用Diagram Viewer, 我们创建了一个测试辅助类RestRequestFor-DiagramViewer用来启动一个请假流程, 见代码清单20-18所示。

代码清单20-18 RestRequestForDiagramViewer.java

```
public class RestRequestForDiagramViewer {
  private static final String BASE_REST_URI =
    "http://localhost:8080/chapter20-rest/service/";

  private static ObjectMapper objectMapper = new ObjectMapper();
  private static boolean formatOutputJson = true;
  public static void main(String[] args) throws IOException {
    // 部署流程
    deployLeave();
    // 查询deployment
    queryDeployment();
    // 查询请假流程的最新版ID
    JsonNode jsonNode = queryLastVersionOfLeaveProcess();
    ArrayNode data = (ArrayNode) jsonNode.get("data");
    JsonNode next = data.iterator().next();
    String latestVersionId = next.get("id").asText();
    System.out.println("latestVersionId=" + latestVersionId);
    // 获取流程的启动表单属性
    queryProcessStartForm(latestVersionId);
    // 启动流程
    jsonNode = startProcessInstance(latestVersionId);
    String processInstanceId = jsonNode.get("id").asText();
    System.out.println("启动了流程: " + processInstanceId);
  }
  // 以上方法的实现代码参见20.5节
}
```

启动chapter20-rest项目后, 打开终端命令行, 执行命令mvn antrun:run-Pinit-db初始化用户数据, 然后运行代码清单20-18的示例代码, 执行后成功启动了一个流程实例, 输出结果如下:

```
=== 启动请假流程 ===
{
  "id" : "5",
  "url" : "http://localhost:8080/chapter20-rest/service/runtime/process-instances/5",
  "businessKey" : null,
  "suspended" : false,
  "processDefinitionId" : "leave:1:4",
  "processDefinitionUrl" : "http://localhost:8080/chapter20-rest/service/repository/process-definitions/leave%3A1%3A4",
  "activityId" : "deptLeaderAudit",
  "variables" : [ ],
  "tenantId" : ""
}
```

观察上面的输出结果可以看出: 成功启动了一个ID为5的流程实例, 关联的流程定义ID为 "leave:1:4" 。结合流程图可以知道流程实例的当前节点应该是 “部门领导审批”, 现在可以在浏览器中访问Diagram Viewer组件来查看流程跟踪图了。

在通过Diagram Viewer组件查看流程图之前先要更改一下该组件所需Rest服务的地址, 打开diagrame-viewer/index.html文件, 找到 “ActivitiRest.options” 代码片段处, 相关代码如下:

```
ActivitiRest.options = {
  processInstanceHighLightsUrl: baseUrl +
    "/service/process-instance/{processInstanceId}
    /highlights?callback=?",
```

```

processDefinitionUrl: baseUrl +
"/service/process-definition/{processDefinitionId}
/diagram-layout?callback=?",
processDefinitionByKeyUrl: baseUrl +
"/service/process-definition/{processDefinitionKey}
/diagram-layout?callback=?",
};

```

把以上代码片段中的“service”更改为本章示例代码web.xml文件中映射路径为“trace”的Servlet，然后在浏览器中打开链接<http://localhost:8080/chapter20-rest/diagram-viewer/index.html?processDefinitionId=leave:1:4&processInstanceId=5>，展示效果如图20-8所示。

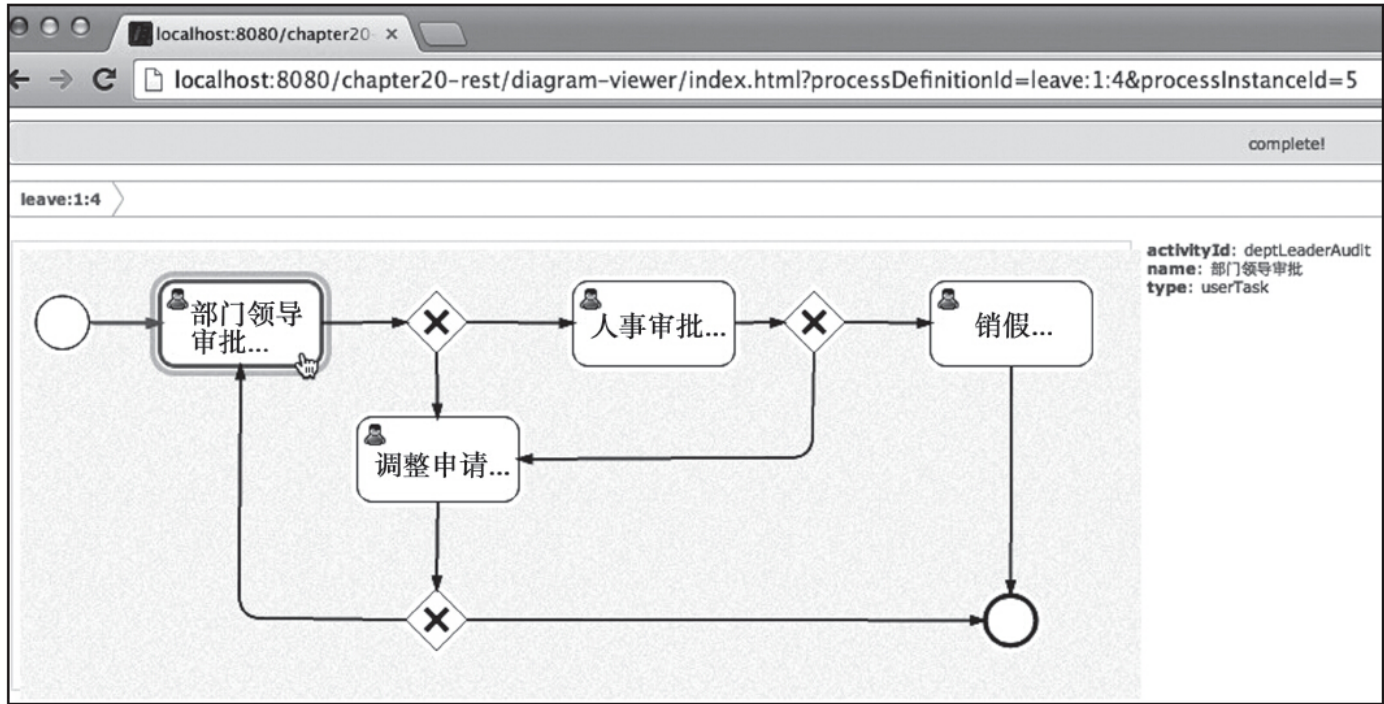


图20-8 访问Diagram Viewer跟踪流程

在图20-8中可以看出，“部门领导审批”任务由红色边框显示（由于黑白印刷，颜色不显示），当鼠标浮动到每一个任务上时，右侧都会展示该任务的一些属性。由于Diagram Viewer完全基于Javascript实现，因此可以根据自己公司的实际需求在Diagram Viewer的基础上改造扩展功能。

20.7 基于REST服务搭建流程中心

在Activiti官网首页中有“Activiti BPM Platform”这样的宣传语，这很明确地表述了Activiti不仅仅是一个流程引擎，还是一个BPM平台，提供的Activiti Rest模块也恰好为BPM平台提供服务，因此可以把Activiti作为BPM平台的基础，使实施者可以很轻松地发布跨平台、跨语言的统一服务接口。

流程中心是很多中大型公司不可缺少的一个公共服务。随着公司规模不断扩大，对业务系统数量的需求也随之增加，为了实现对各个业务系统的统一管理，对各个系统的流程使用统一规范，流程中心理所当然成为必备的服务。

在本节笔者根据一些经验简单地谈谈流程中心的一些架构以及实施办法，希望对读者有些帮助。

20.7.1 基础架构

图20-9展示了一张简单的架构图，描述了业务系统与流程中心的基础架构与交互过程。

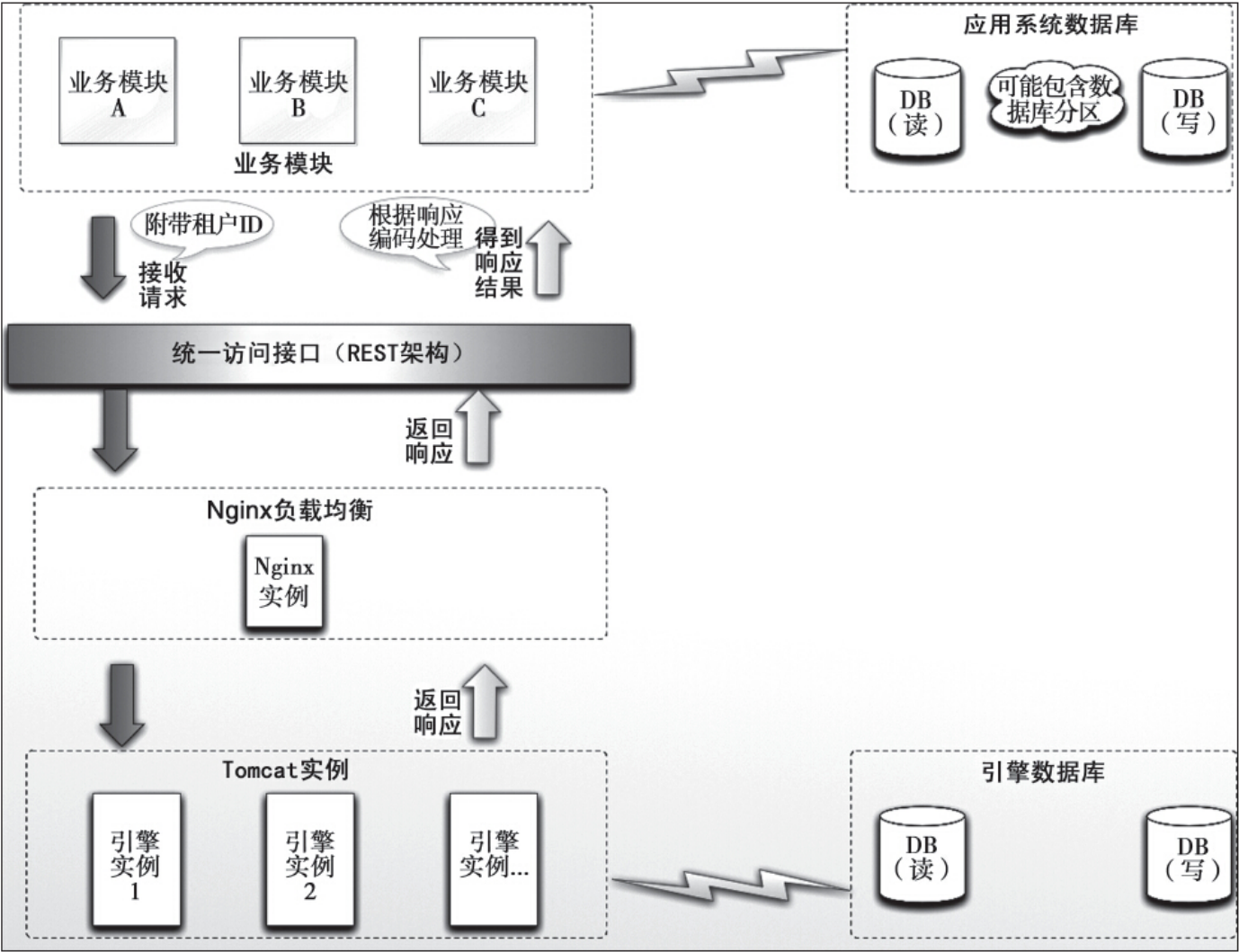


图20-9 基础架构图

图20-9使用的是单一数据库架构，随着业务系统的增多，当单一数据库不能满足需求时，可以考虑根据业务系统分割数据库，也就是每一个业务系统拥有一套Activiti引擎数据库，并根据数据库实例的数量创建多个引擎对象，在每一个引擎对象中配置数据库的别名来标示数据库的名称，这样就可以实现一个引擎管理多套数据库。关于数据库别名的配置如下：

```
<bean id="processEngineConfiguration"
      class="org.activiti.xxx.ProcessEngineConfiguration">
  <property name="databaseTablePrefix" value="DB_1" />
</bean>
```

20.7.2 表单模式选型

流程中心的一大特点就是“通用”，所以选择表单模式时要选择支持不同的业务系统。经综合考虑笔者认为选择“外置表单”的模式最佳，但是与本书第6章介绍的外置表单使用方式有些不同，虽然各个业务系统的开发人员在设计流程时仍然使用“`activiti:formKey`”属性配置每个流程中与任务关联的表单，但是设置的值不是表单文件的名称（xxx.form），而是一个相对地址的URL（每个接入系统的基础URL可以保存在流程中心的数据库表中），而由“`activiti:formKey`”属性配置除基础URL之外的相对路径，这样各个业务系统在通过REST接口获取任务表单属性后就可以从当前业务系统中获取表单内容。图20-10展示了一个大概的表单交互过程。

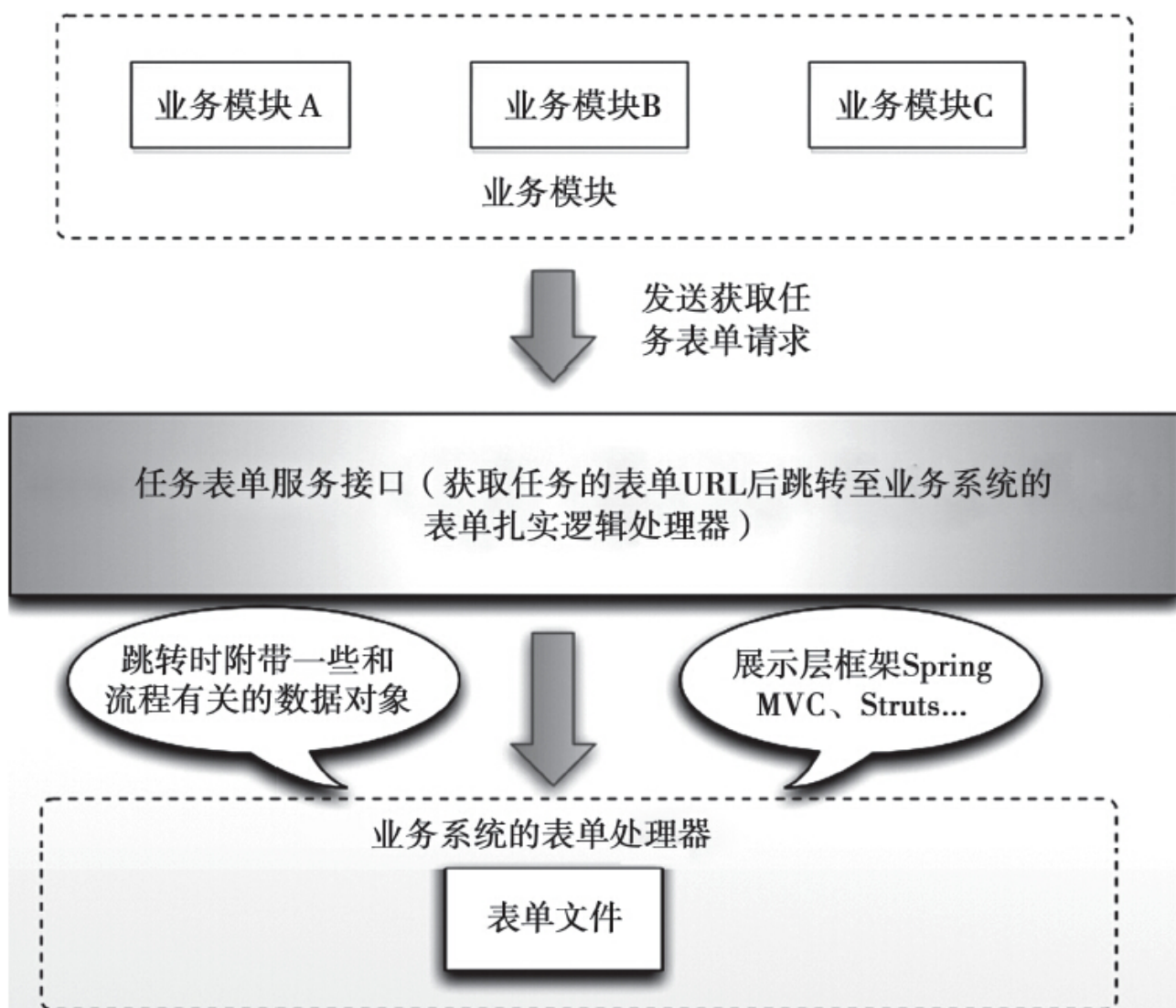


图20-10 流程中心处理业务系统表单的过程

20.7.3 统一的组件

除了和流程引擎使用统一的REST服务之外，流程中心还提供一些通用的组件，例如20.6节介绍的流程图跟踪组件Diagram Viewer。如果有统一的UI组件，还可以封装一些通用功能，例如通用的按钮动作处理：提交、转办、委派，或者批注、事件等。

对于查询待办任务、运行中流程、已结束流程等数据，这种通用的操作应该由流程中心提供，业务系统只需要关心业务逻辑，可以考虑把所有业务系统的流程数据在一个系统中展示（例如Portal系统），也可以在每一个业务系统中嵌入由流程中心封装过的流程数据查询模块。

由于业务数据与流程数据分离，当查询用户的待办任务时会遇到数据难以过滤的问题，因为流程中没有保存业务数据，如果待办任务较多，用户会对这些待办任务无从下手，所以笔者在设计流程中心时加入了对待办任务查询的可配置功能。

在启动流程或任务中更新了业务数据后用变量的方式把业务字段记录到引擎表，然后再为每一个流程配置可用的查询字段，在待办任务列表中选择不同的流程切换相关的业务查询字段，这样用户就可以切换不同流程的待办任务且可以使用业务数据过滤结果。

20.7.4 事务管理

第7章介绍的Spring代理流程引擎，它的一个最大好处在于可以由Spring统一管理Activiti引擎与业务系统的事务。而REST服务是独立运行的，为了实现统一的事务管理可以选择使用分布式事务管理，不过这个方案在企业内部用得不多。笔者倾向于选择另外一种稍“弱”的通用事务管理方式，由业务方判断REST服务的状态码决定业务系统事务的“提交”与“回滚”，例如，启动流程成功后REST服务返回的状态码为“201”，启动失败后返回的是“400”。

首先业务系统把用户填写的表单数据保存（此时事务部提交，保持开启状态），然后请求REST服务根据返回的状态码与JSON数据判断流程中心是否正确处理了请求，如果处理成功，则提交业务系统的事务，否则执行回滚动作。

20.8 集成流程设计器Activiti Modeler

在3.2节中介绍了如何在Activiti Explorer中使用Activiti Modeler流程设计器，由于Activiti Modeler依赖REST服务模块，故本节将介绍如何集成Activiti Modeler组件。

集成Modeler与集成Diagram Viewer的步骤类似，不过步骤略显复杂（笔者也希望以后官方能够简化集成的步骤，因为Modeler相关的文件过于分散）。

Activiti Modeler涉及一个概念——模型（Model），模型也是Activiti Modeler流程设计器的产物，并且模型与流程定义（Process Definition）两者可以互相转换，大致的转换过程可以参考图20-

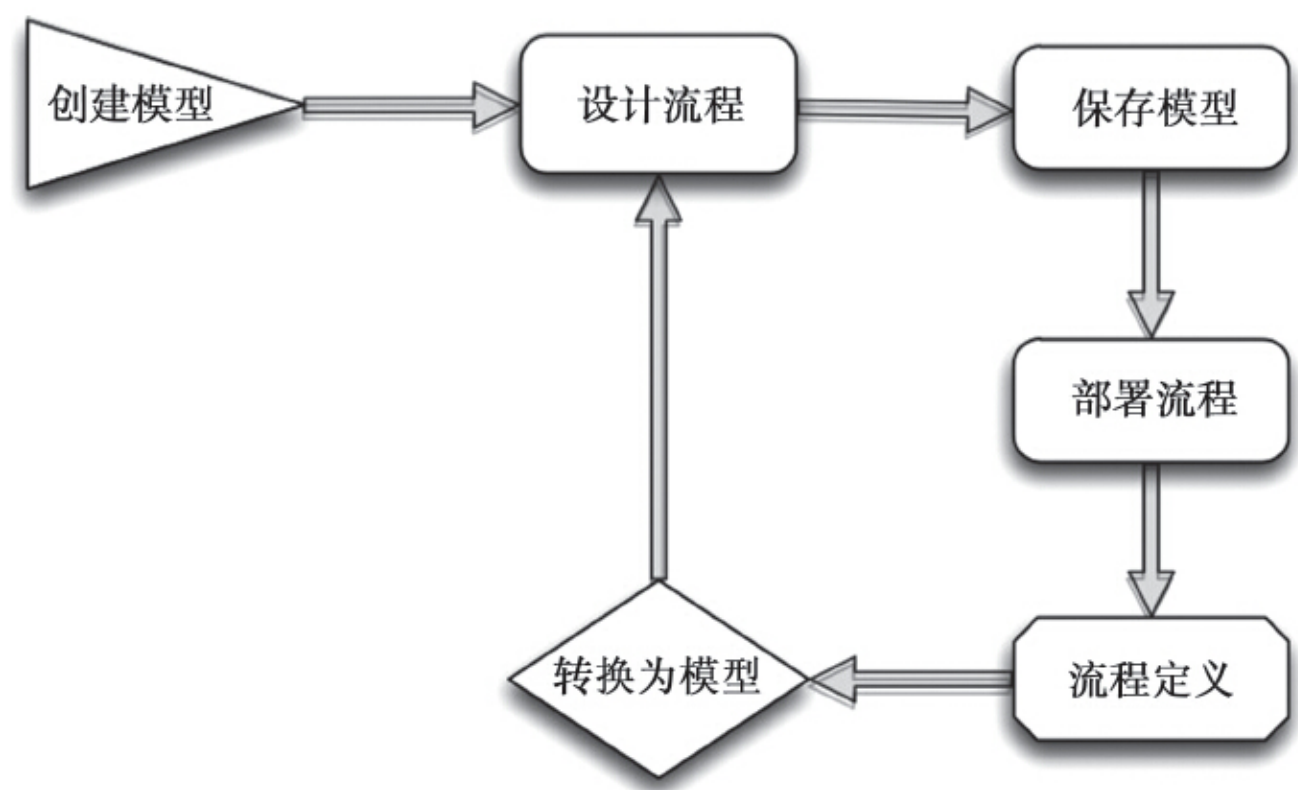


图20-11 Activiti Modeler的模型与流程定义的转换过程

20.8.1 准备资源文件

解压Activiti官方提供的activiti-explorer.war后可以看到如图20-12所示的目录结构，图20-13则展示了classes目录的结构。

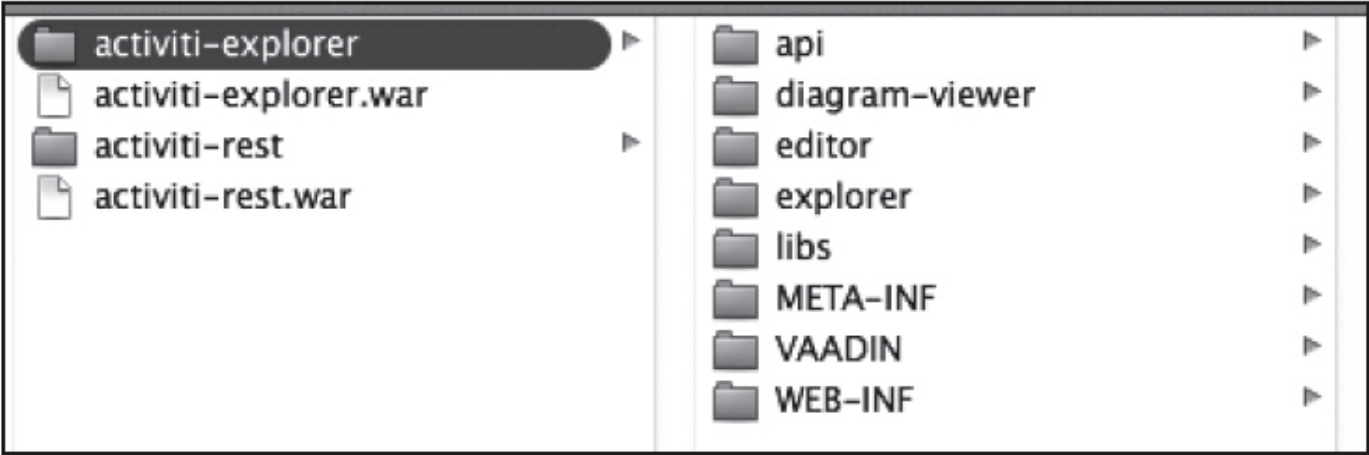


图20-12 activiti-explorer的目录结构

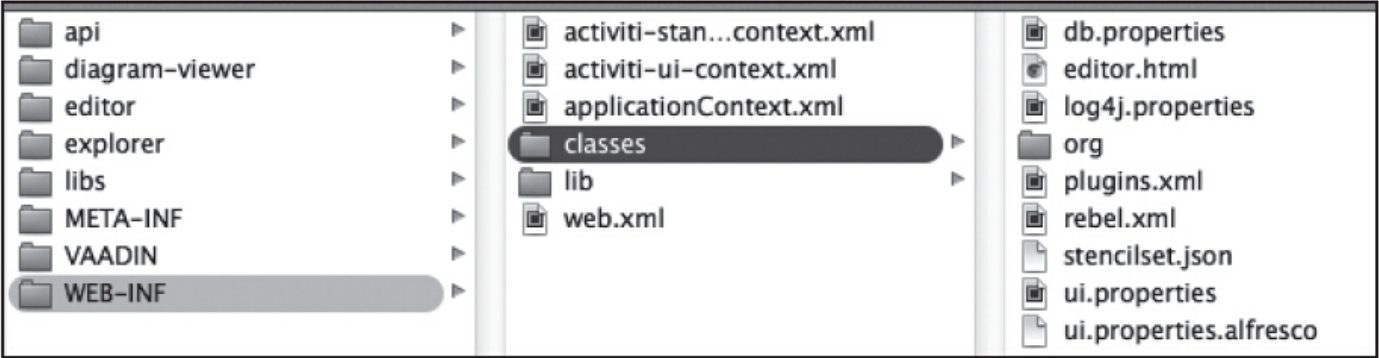


图20-13 activiti-explorer的classes目录结构

把图20-12中所示的api、editor、explorer、libs目录复制到本章示例项目的chapter20-rest/src/main/webapp目录中，然后把图20-13中所示的editor.html、stencilset.json、plugins.xml复制到chapter20-rest/src/main/resources目录中，最终的目录结构如图20-14所示。

▼ chapter20-rest

▼ src

▼ main

▶ java

▼ resources

▶ data

▶ diagrams

 activiti-context.xml

 db.properties

 editor.html

 log4j.properties

 plugins.xml

 stencilset.json

▼ webapp

▶ api

▶ common

▶ diagram-viewer

▶ editor

▶ explorer

▶ libs

▶ static

▶ WEB-INF

 index.jsp

图20-14 复制了Activiti Modeler相关文件后的目录结构

20.8.2 准备配置文件

资源文件复制好后需要配置Activiti Modeler访问的REST服务路由，与配置Diagram Viewer的REST服务类似，在web.xml中添加Modeler模块的REST路由配置，相关配置代码如下：

```
<!-- Modeler REST服务 -->
<servlet>
  <servlet-name>ModelerRestApplication    </servlet-name>
  <servlet-class>org.restlet.ext.servlet.ServerServlet</servlet-class>
  <init-param>
    <!-- Application class name -->
    <param-name>org.restlet.application    </param-name>
    <param-value>
      org.activiti.rest.editor.application.ModelerRestApplication
    </param-value>
  </init-param>
</servlet>
<servlet-mapping>
  <servlet-name>ModelerRestApplication</servlet-name>
  <url-pattern>/mservice/*</url-pattern>
</servlet-mapping>
```

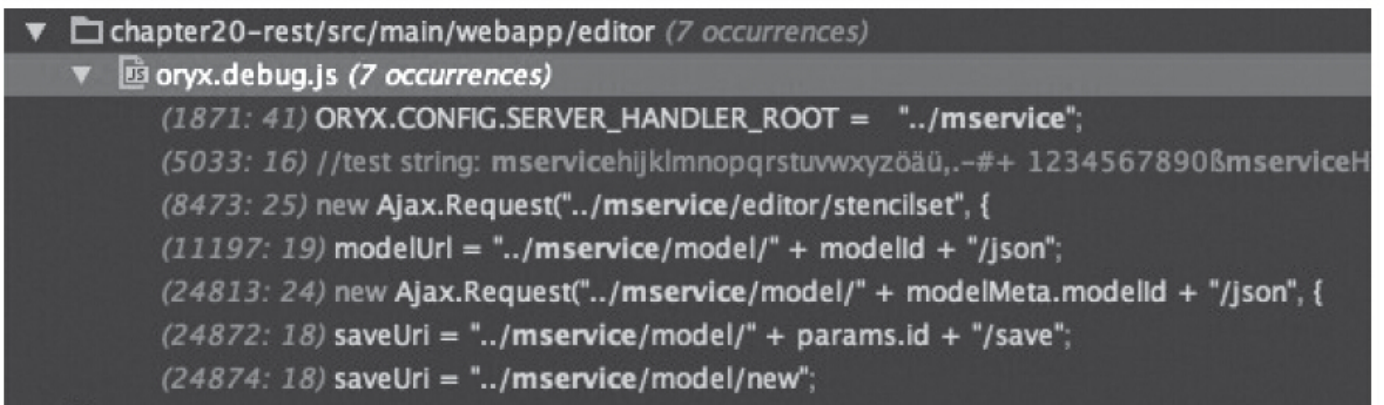
以上配置中的ModelerRestApplication类被映射到路径“/mservice/*”上（表示Modeler Service），该路由类位于activiti-modeler模块中，所以还需要在pom.xml中添加相关的依赖配置（或者jar包），依赖配置如下：

```
<dependency>
  <groupId>org.activiti</groupId>
  <artifactId>activiti-modeler</artifactId>
  <version>${activiti-version}</version>
</dependency>
```

20.8.3 更改默认配置

由于Activiti Modeler的一些资源文件从Activiti Explorer中复制而来，所以有关REST服务的配置需要做一些调整。

编辑文件chapter20-rest/src/main/webapp/editor/oryx.debug.js，替换其中的“/service/”为“/mservice/”，从图20-15中的搜索结构可以清晰地看出和mservice相关的配置更改。



```
(1871: 41) ORYX.CONFIG.SERVER_HANDLER_ROOT = "../mservice";
(5033: 16) //test string: mservicehijklmnopqrstuvwxyzöäü,.-#+ 1234567890ßmserviceH
(8473: 25) new Ajax.Request("../mservice/editor/stencilset", {
(11197: 19) modelUrl = "../mservice/model/" + modelId + "/json";
(24813: 24) new Ajax.Request("../mservice/model/" + modelMeta.modelId + "/json", {
(24872: 18) saveUri = "../mservice/model/" + params.id + "/save";
(24874: 18) saveUri = "../mservice/model/new";
```

图20-15 oryx.debug.js文件中需要被替换的相关代码

20.8.4 创建模型

创建模型也就是调用引擎的API创建一个Model对象。可以查看引擎的API文档RepositoryService接口中有关Model的一些方法，如图20-16所示。

newModel()

Creates a new model.

saveModel(Model model)

Saves the model.

图20-16 RepositoryService中有关模型的API

图20-17展示了通过用户界面创建模型对象，在输入三个参数后单击“创建”按钮即可创建一个模型，随即打开如图20-18所示的编辑模型界面，从左侧“组件库”中拖动组件设计好流程，然后单击图20-18中左上角的保存按钮，打开如图20-19所示的模型列表页面即可看到已经保存的模型。

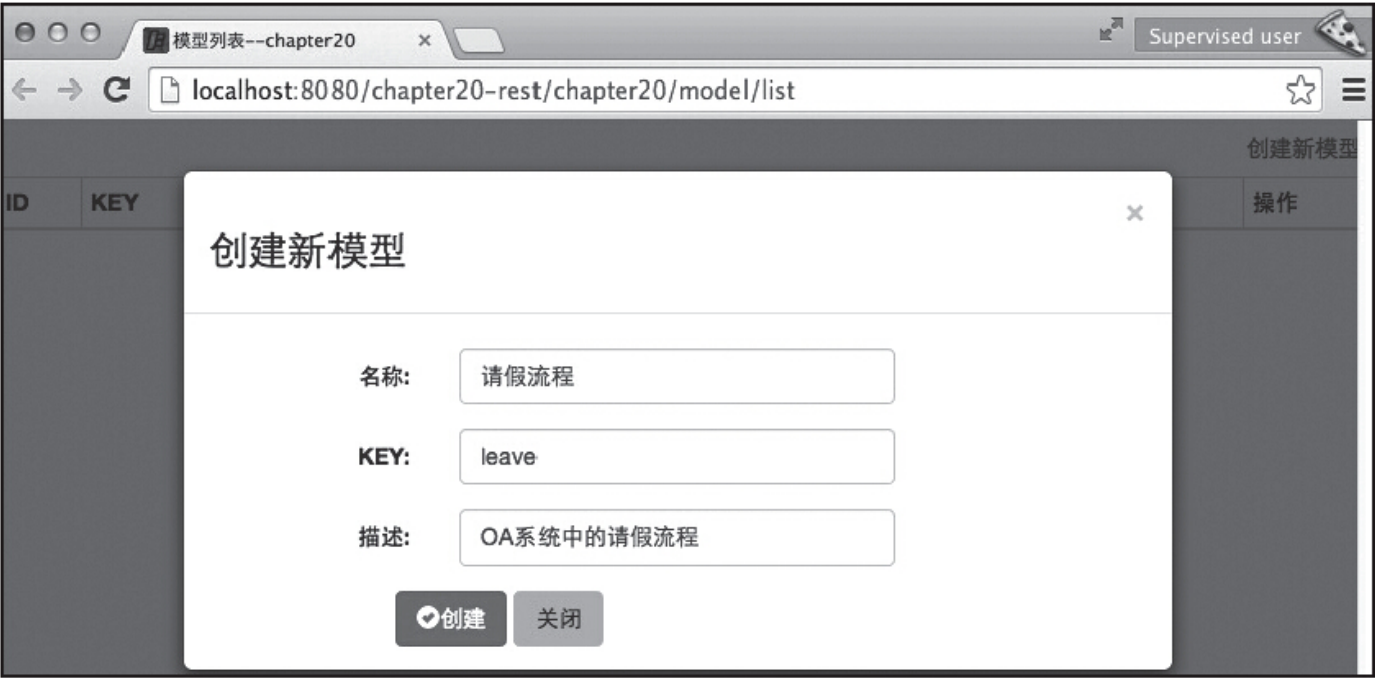


图20-17 创建新模型对话框

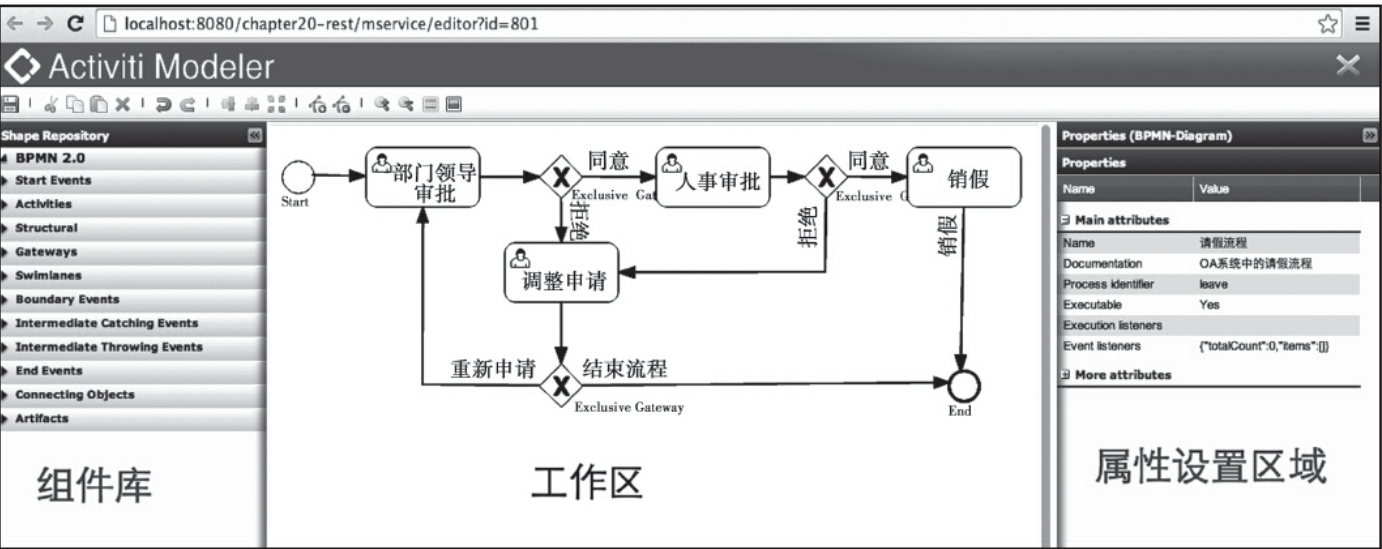


图20-18 编辑模型界面



图20-19 模型列表

图20-17、图20-18和图20-19的相关代码可以在本章示例代码的src/main/java/me/kafeitu/activiti/web/chapter20/ModelController.java以及src/main/webapp/WEB-INF/views/chapter20/model-list.jsp中找到，其中ModelerController的映射路径为“/chapter20/model”。

图20-17的创建新模型的代码见代码清单20-19。

代码清单20-19 创建新模型的实现代码

```
@RequestMapping(value = "create")
public void create(@RequestParam("name") String name, // 模型名称
                  @RequestParam("key") String key, // 模型KEY
                  @RequestParam(value = "description", required = false) String description,
                  HttpServletRequest request, HttpServletResponse response) {
    try {
        ObjectMapper objectMapper = new ObjectMapper();
        ObjectNode modelObjectNode = objectMapper.createObjectNode(); // #1-S
        modelObjectNode.put(ModelDataJsonConstants.MODEL_NAME, name);
        modelObjectNode.put(ModelDataJsonConstants.MODEL_REVISION, 1);
        modelObjectNode.put(ModelDataJsonConstants
            .MODEL_DESCRIPTION, StringUtils.defaultString(description)); // #1-E
        Model newModel = repositoryService.newModel(); // 创建模型对象 // #2
        newModel.setMetaInfo(modelObjectNode.toString()); // #3-S
        newModel.setName(name);
        newModel.setKey(StringUtils.defaultString(key)); // #3-E
        ObjectNode editorNode = objectMapper.createObjectNode(); // #4-S
        editorNode.put("id", "canvas");
        editorNode.put("resourceId", "canvas");
        ObjectNode stencilSetNode = objectMapper.createObjectNode();
        stencilSetNode.put("namespace", "http://b3mn.org/stencilset/bpmn2.0#");
        editorNode.put("stencilset", stencilSetNode);
        // 为模型绑定参数
        repositoryService.addModelEditorSource(newModel.getId(), // #4-E
            editorNode.toString().getBytes("utf-8"));
        // 打开模型设计界面
        response.sendRedirect(request.getContextPath() + // #5
            "/mservice/editor?id=" + newModel.getId());
    } catch (Exception e) {
        logger.error("创建模型失败: ", e);
    }
}
```

在代码清单20-19的#2处创建一个新的模型对象，在#3处将#1处定义的模型的元信息（Meta Info）设置到新模型对象中并保存新模型对象到引擎数据库（表ACT_RE_MODEL），在#4处为新模型设置设计器相关属性（以二进制的方式保存在ACT_GE_BYTEARRAY表中，通过字段EDITOR_SOURCE_VALUE_ID字段关联，数据为JSON格式），最后在#5处把相应结果重定向到activiti-modeler模块的模型设计界面服务，展示如图20-18所示的模型编辑界面。

20.8.5 导出模型的流程XML

单击图20-19中的“导出”链接即可下载一个XML格式的文件。该文件是一个标准的流程文件，导出的文件内容如图20-20所示，相关代码见代码清单20-20所示。

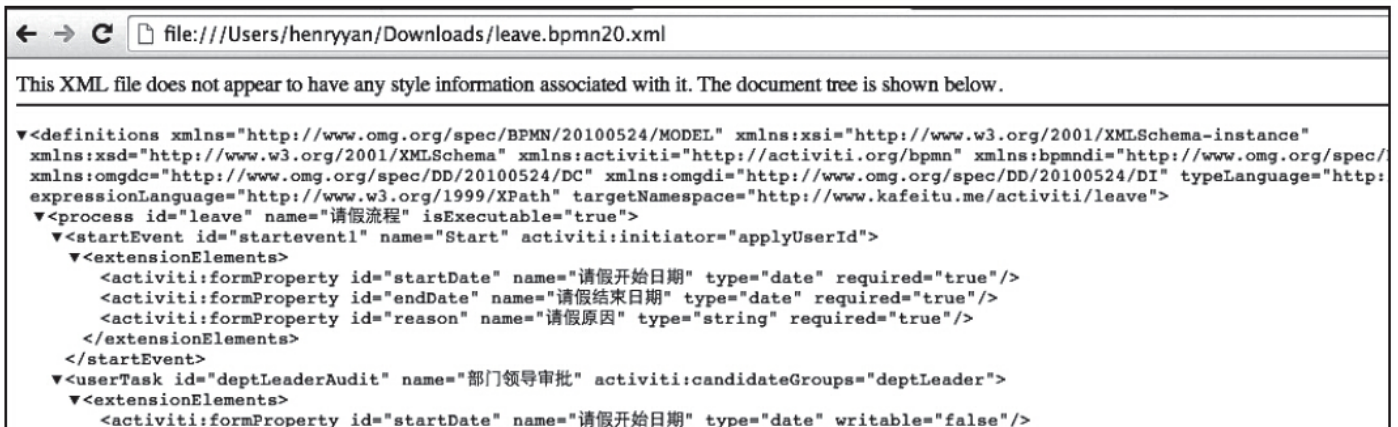


图20-20 从模型中导出的流程文件内容

代码清单20-20 从模型对象中导出流程文件

```
@RequestMapping(value = "export/{modelId}")
public void export(@PathVariable("modelId") String modelId, HttpServletResponse response) {
    try {
        Model modelData = repositoryService.getModel(modelId); // 获取模型对象
        BpmnJsonConverter jsonConverter = new BpmnJsonConverter();
        JsonNode editorNode = new ObjectMapper() // 从模型对象中读取JSON数据
            .readTree(repositoryService.getModelEditorSource(modelData.getId()));
        // 把JSON对象转换为BpmnModel
        BpmnModel bpmnModel = jsonConverter.convertToBpmnModel(editorNode);
        BpmnXMLConverter xmlConverter = new BpmnXMLConverter();
        // 把BpmnModel对象转换成XML格式
        byte[] bpmnBytes = xmlConverter.convertToXML(bpmnModel);
        // 创建输入流输出文件内容到浏览器
        ByteArrayInputStream in = new ByteArrayInputStream(bpmnBytes);
        IOUtils.copy(in, response.getOutputStream());
        String filename = bpmnModel.getMainProcess().getId() + ".bpmn20.xml";
        response.setHeader("Content-Disposition", "attachment; filename=" + filename);
        response.flushBuffer();
    } catch (Exception e) {
        logger.error("导出model的xml文件失败: modelId={}", modelId, e);
    }
}
```

20.8.6 把模型转换为流程定义

单击图20-19中的“部署”链接即可把模型转换成流程定义并部署到引擎中，如图20-21所示。相关代码见代码清单20-21所示。



图20-21 把模型转换成流程定义后的流程定义列表

代码清单20-21 把模型转换为流程定义并部署

```
@RequestMapping(value = "deploy/{modelId}")
public String deploy(@PathVariable("modelId") String modelId, RedirectAttributes redirectAttributes) {
    try {
        Model modelData = repositoryService.getModel(modelId); // #1-S
        ObjectNode modelNode = (ObjectNode) new ObjectMapper()
            .readTree(repositoryService.getModelEditorSource(modelData.getId()));
        byte[] bpmnBytes = null;
        BpmnModel model = new BpmnJsonConverter().convertToBpmnModel(modelNode); // #1-E
        bpmnBytes = new BpmnXMLConverter().convertToXML(model); // #1-E
        String processName = modelData.getName() + ".bpmn20.xml";
        Deployment deployment = repositoryService.createDeployment() // #2-S
            .name(modelData.getName()) // 流程名称
            .addString(processName, new String(bpmnBytes)) // 以String方式部署
            .deploy(); // 部署 // #2-E
    } catch (Exception e) {
        logger.error("根据模型部署流程失败: modelId={}", modelId, e);
    }
    return "redirect:/chapter20/model/list";
}
```

代码清单20-21中，#1处的代码与代码清单20-20中重合，#2处创建了Deployment对象并把模型的流程XML字节转换为String类型部署到引擎中。

20.8.7 把流程定义转换为模型

图20-22描述了模型与流程定义的互转过程，我们可以把某一个版本的流程定义转换成模型对象，在完成模型设计后再部署到引擎中，在实际应用中会经常以这种方式迭代修改流程逻辑。

单击图20-22中所显示的“转换为模型”选项，页面跳转到模型列表页面，从图20-23中可以看出多出了一个模型记录。相关的代码见代码清单20-22所示。



图20-22 把流程定义转换为模型

←

→

↺

localhost:8080/chapter20-rest/chapter20/model/list

☆

☰

创建新模型

ID	KEY	名称	版本	创建时间	最后更新时间	元数据	操作
801	leave	请假流程	1	Sat Jun 28 17:56:59 CST 2014	Sat Jun 28 18:01:18 CST 2014	{"name":"请假流程","revision":1,"description":"OA系统中的请假流程"}	编辑 部署 导出 删除
807	leave	请假流程.bpmn20.xml	1	Sat Jun 28 19:17:30 CST 2014	Sat Jun 28 19:17:30 CST 2014	{"name":"请假流程","revision":1,"description":null}	编辑 部署 导出 删除

共 2 条<<<1>>>

图20-23 转换流程定义为模型后的模型列表

代码清单20-22 把流程定义转换为模型

```
@RequestMapping(value = "/process/convert-to-model/{processDefinitionId}")
public String convertToModel(
    @PathVariable("processDefinitionId") String processDefinitionId)
    throws UnsupportedEncodingException, XMLStreamException {
    // 查询流程定义
    ProcessDefinition processDefinition = repositoryService.createProcessDefinitionQuery()
        .processDefinitionId(processDefinitionId).singleResult();
    // 读取流程定义的XML输入流
    InputStream bpmnStream = repositoryService
        .getResourceAsStream(processDefinition.getDeploymentId(),
            processDefinition.getResourceName());
    XMLInputFactory xif = XMLInputFactory.newInstance(); // #1-S
    InputStreamReader in = new InputStreamReader(bpmnStream, "UTF-8");
    XMLStreamReader xtr = xif.createXMLStreamReader(in);
    BpmnModel bpmnModel = new BpmnXMLConverter().convertToBpmnModel(xtr);
    BpmnJsonConverter converter = new BpmnJsonConverter();
    ObjectNode modelNode = converter.convertToJson(bpmnModel);
    Model modelData = repositoryService.newModel(); // #1-L
    modelData.setKey(processDefinition.getKey());
    modelData.setName(processDefinition.getResourceName());
    modelData.setCategory(processDefinition.getDeploymentId());
    ObjectNode modelObjectNode = new ObjectMapper().createObjectNode();
    modelObjectNode.put(ModelDataJsonConstants.MODEL_NAME,
        processDefinition.getName());
    modelObjectNode.put(ModelDataJsonConstants.MODEL_REVISION, 1);
    modelObjectNode.put(ModelDataJsonConstants.MODEL_DESCRIPTION,
        processDefinition.getDescription());
    modelData.setMetaInfo(modelObjectNode.toString()); // #1-E
    // 保存模型并设置模型的属性
    repositoryService.saveModel(modelData);
    repositoryService.addModelEditorSource(modelData.getId(), modelNode.toString().getBytes("utf-8"));
    return "redirect:/chapter20/model/list";
}
```

代码清单20-22综合了本节已经涉及的功能，#1处设置了一些模型的属性并在#1-1处创建一个新的模型，最后保存模型对象，过程综合了模型导出XML和创建新模型。

20.8.8 删除模型

单击图20-19中所示的“删除”链接可以删除一个模型记录，相关代码见代码清单20-23所示。

代码清单20-23 删除模型

```
@RequestMapping(value = "delete/{modelId}")
public String delete(@PathVariable("modelId") String modelId) {
    repositoryService.deleteModel(modelId);
    return "redirect:/chapter20/model/list";
}
```

20.9 本章小结

本章详细介绍了Activiti REST架构的API，它为访问引擎提供了另外一种交互方式，有利于分布式服务的搭建。另外，还介绍了如何使用不同的客户端访问REST API，并且通过一个完整的示例展示了一个流程的生命周期。

REST API的最大目的是搭建一个流程中心（或平台），所以Activiti官方提供了基于B/S架构的统一的流程跟踪组件——Diagram Viewer，以及流程设计器——Activiti Modeler，并从架构的层面阐述了如何基于REST搭建通用的流程中心。

第21章 入侵Activiti

本章应该算高级篇中最“高级”的内容，因为本章内容大多与Activiti引擎的内部实现有关的，剖析了一些技术的实现原理，以及如何更改默认实现或添加自定义实现从而“侵入”到引擎内部。当默认

实现不能满足我们的需求时，这项侵入引擎的“技能”就显得尤为重要了。学习完本章内容后相信强大的你可以用“透明”的方式看穿每一个动作的执行过程。

21.1 解析BPMN文件

回顾一下图1-1展示的流程生命周期图，开发人员根据业务人员的需求设计完流程定义后就要审查部署到引擎验证设计的流程是否符合预期目标，在Activiti中“部署”动作总是排在第一步进行的，引擎会解析XML格式的流程文件提取流程定义（ProcessDefinition）对象，部署动作由BpmnDeployer类负责图21-1展示了该类依赖的各种解析、处理器的结构图。从图21-1中可以看到很多种处理器，每一种处理器下面还有具体的活动解析实现类。最终处理完成后会得到（一个或者多个）流程定义对象，之后再把这些对象保存到引擎数据库，同时也会把解析后得到的流程定义对象缓存起来便于后续使用。

图21-2展示了一个XML格式的流程文件如何经过几个大的步骤部署到引擎的过程。

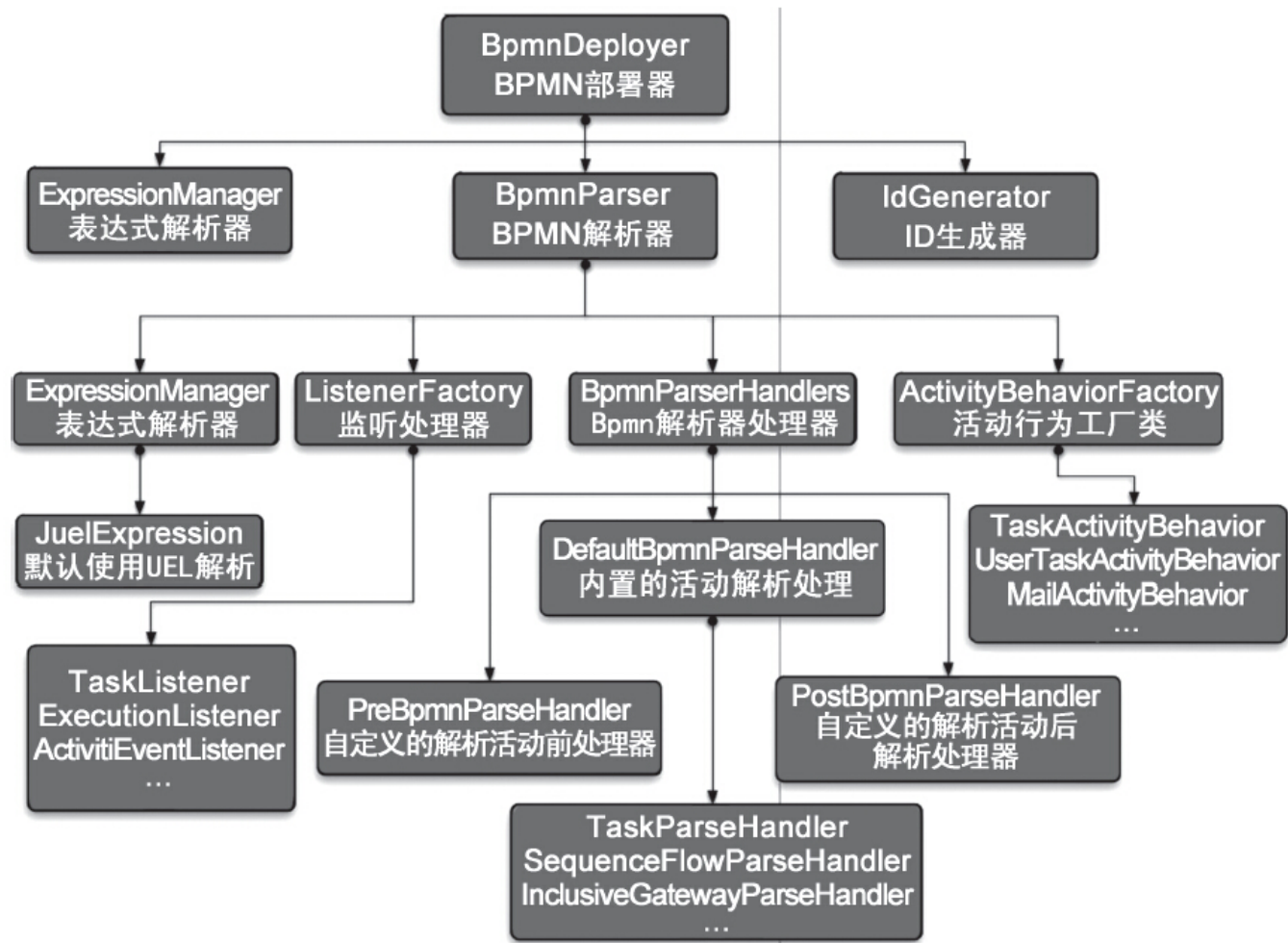


图21-1 部署流程的处理类BpmnDeployer的依赖图

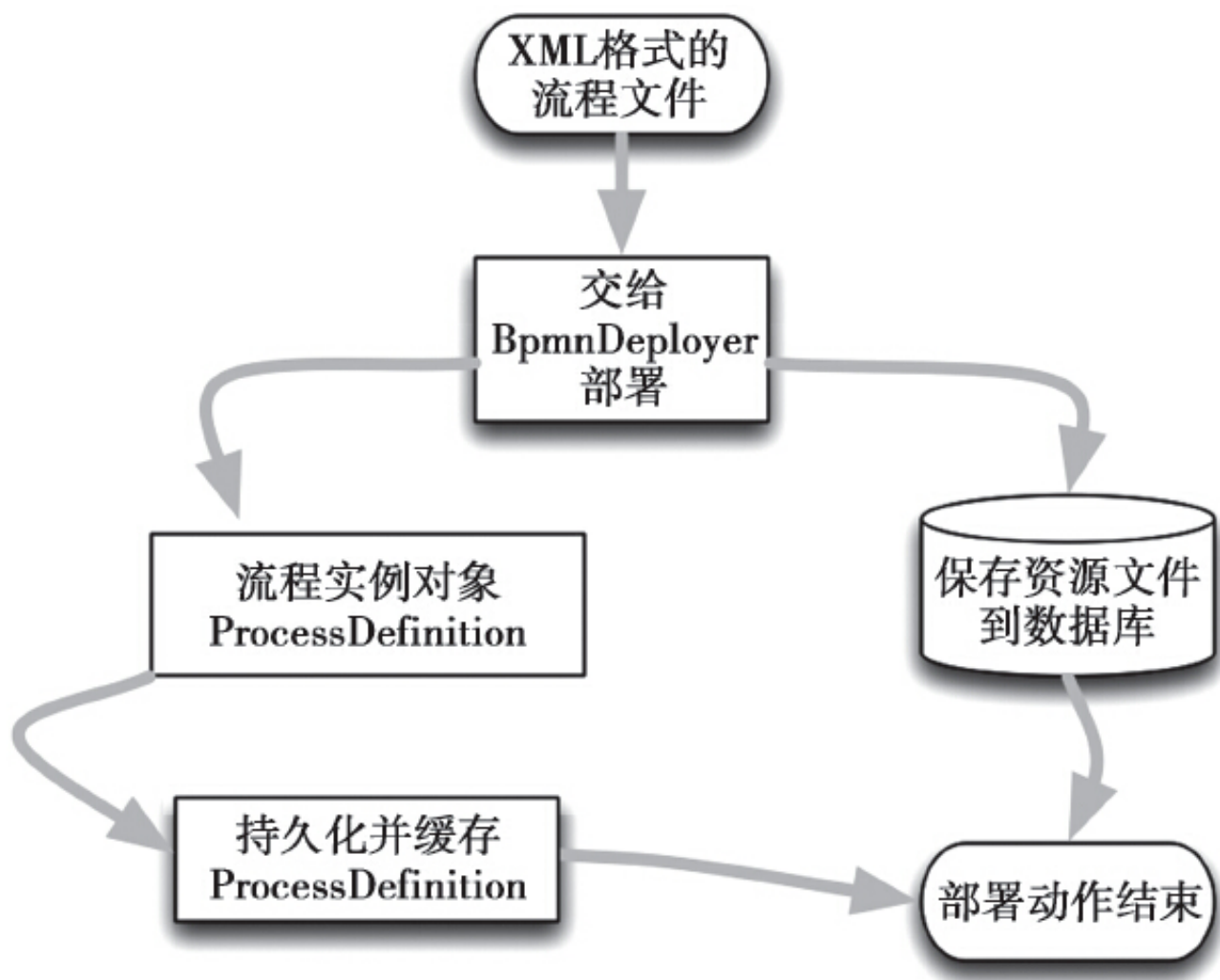


图21-2 流程文件部署到引擎的大致过程

21.1.1 BpmnModel对象与XML之间的转换

BpmnModel对象是Activiti中用来描述在流程中组建的一个普通Bean，包含了流程文件中所有活动（Activity）的定义，打开该类的源码，从中可以看到下面的属性定义：

```
public class BpmnModel {
    // definitions的一些属性集合
    protected Map<String, List<ExtensionAttribute>>
    definitionsAttributes = new LinkedHashMap<>();
    // 流程文件中定义的流程集合（一个流程文件中可以定义多个流程）
    protected List<Process> processes = new ArrayList<>();
    ...
    // getter and setter
}
```

在13.3节中就已经介绍过如何获取BpmnModel对象，然后根据BpmnModel对象再生成流程图。获取BpmnModel对象的相关代码如下：

```
BpmnModel bpmnModel = repositoryService
    .getBpmnModel(processDefinitionId);
```

以上代码使用了Activiti的activiti-bpmn-converter模块提供的BpmnModel对象与XML的互转功能，通过创建org.activiti.bpmn.converter.BpmnXMLConverter类对象调用相应的方法即可实现BpmnModel对象与XML之间的转换操作。下面通过代码清单21-1和代码清单21-2分别演示两者之间互相转换的API。

不过在学习之前要先在Maven配置文件pom.xml中添加如下的依赖配置，BpmnModel类位于模块activiti-bpmn-model中，BpmnXMLConverter类位于activiti-bpmn-converter模块中。

```
<dependency>
    <groupId>org.activiti</groupId>
    <artifactId>activiti-bpmn-model</artifactId>
    <version>${activiti.version}</version>
</dependency>
<dependency>
    <groupId>org.activiti</groupId>
    <artifactId>activiti-bpmn-converter</artifactId>
    <version>${activiti.version}</version>
</dependency>
```

代码清单21-1演示了如何把一个XML文件内容转换成BpmnModel对象并验证转换后得到的BpmnModel对象中是否包含了流程（Process）对象。

 **提示** 到底BpmnModel对象中包含了哪些属性，可以在IDE中用Debug断点查看该对象中的所有属性，由于篇幅有限本书就不详细介绍了。

代码清单21-1 把XML内容转换成BpmnModel对象

```
public class BpmnModelTest extends AbstractTest {
    // 把XML转换成BpmnModel对象
    @Test // 测试时自动部署流程
    @Deployment(resources = "chapter6/dynamic-form/leave.bpmn")
```

```

public void testXmlToBpmnModel() throws Exception {
    // 根据流程定义获取XML资源文件流对象
    ProcessDefinition processDefinition = repositoryService    #1-S
        .createProcessDefinitionQuery()
        .processDefinitionKey("leave").singleResult();
    String resourceName = processDefinition.getResourceName();
    InputStream inputStream = repositoryService
        .getResourceAsStream(processDefinition.getDeploymentId(), resourceName);    #1-E
    // 创建转换对象
    BpmnXMLConverter converter = new BpmnXMLConverter();    #2
    // 创建XMLStreamReader读取XML资源
    XMLInputFactory factory = XMLInputFactory.newInstance();
    XMLStreamReader reader = factory.createXMLStreamReader(inputStream);
    // 把XML转换成BpmnModel对象
    BpmnModel bpmnModel = converter.convertToBpmnModel(reader);    #3
    // 验证BpmnModel对象不为空
    assertNotNull(bpmnModel);
    Process process = bpmnModel.getMainProcess();
    assertEquals("leave", process.getId()); // 验证转换得到的流程ID
}
}

```

代码清单21-1的#1处从引擎中读取流程定义并根据XML资源文件名称获取输入流对象，#2处创建了一个BpmnXMLConverter类的实例，#3处调用convertToBpmnModel()方法得到一个BpmnModel对象并在最后验证了该对象中是否包含了流程（Process）对象。

其中#1处读取Bpmn文件输入流也可以更改成从Classpath读取，代码如下：

```

InputStream inputStream = getClass().getClassLoader()
    .getResourceAsStream("chapter6/dynamic-form/leave.bpmn");

```

与代码清单21-1相反，代码清单21-2展示了如何把BpmnModel对象转换成XML格式的字符串。

代码清单21-2 把BpmnModel对象转换成XML内容

```

public class BpmnModelTest extends AbstractTest {
    @Test
    @Deployment(resources = "chapter6/dynamic-form/leave.bpmn")
    public void testBpmnModelToXml() throws Exception {
        // 查询流程定义对象
        ProcessDefinition processDefinition = repositoryService
            .createProcessDefinitionQuery()
            .processDefinitionKey("leave").singleResult();
        // 获取BpmnModel对象
        BpmnModel bpmnModel = repositoryService
            .getBpmnModel(processDefinition.getId());
        // 创建转换对象
        BpmnXMLConverter converter = new BpmnXMLConverter();
        // 把BpmnModel对象转换成字符（也可以输出到文件中）
        byte[] bytes = converter.convertToXML(bpmnModel);
        String xmlContent = new String(bytes);
        System.out.println(xmlContent);
    }
}

```

执行完代码清单21-2后，在控制台可以看到的输出内容如图21-3所示。

```

11:20:06,947 [main] DEBUG org.activiti.engine.impl.interceptor.LogInterceptor -
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL" xmlns:xsi="http://w
<process id="leave" name="请假流程-动态表单" isExecutable="true">
  <documentation>请假流程演示-动态表单</documentation>
  <startEvent id="startevent1" name="Start" activiti:initiator="applyUserId">
    <extensionElements>
      <activiti:formProperty id="startDate" name="请假开始日期" type="date" datePat
      <activiti:formProperty id="endDate" name="请假结束日期" type="date" datePatter

```

图21-3 把BpmnModel对象转换为XML内容输出

21.1.2 动态创建流程

了解了BpmnModel对象与XML之间的转换，再通过一个有意思的示例了解一下这个转换过程的实际用途：Activiti团队成员“Frederik Heremans”的博客中有一篇博文，介绍的是如何动态创建一个流程并（仅100行代码），读者可以参考原文《Dynamic Process Creation and Deployment in 100Lines of Code》^[1]。这篇博文发表于Activiti 5.12版本发布之前，笔者觉得这能勾起读者的兴趣，所以在这里单独介绍其运作过程，完整代码见代码清单21-3（在原文的基础之上做了一些微调）。

代码清单21-3 动态创建流程

```

public class DynamicProcess {
    @Rule
    public ActivitiRule activitiRule = new ActivitiRule();
    @Test
    public void testDynamicDeploy() throws Exception {
        // 1. 创建一个空的BpmnModel和Process对象    #1
        BpmnModel model = new BpmnModel();
        Process process = new Process();
        model.addProcess(process);
        process.setId("my-process");
        // 创建Flow元素（所有的事件、任务都被认为是Flow）
        process.addFlowElement(createStartEvent());
        process.addFlowElement(createUserTask("task1", "First task", "fred"));
        process.addFlowElement(createUserTask("task2", "Second task", "john"));
        process.addFlowElement(createEndEvent());
        // 创建顺序流
        process.addFlowElement(createSequenceFlow("start", "task1"));
        process.addFlowElement(createSequenceFlow("task1", "task2"));
        process.addFlowElement(createSequenceFlow("task2", "end"));
        // 2. 流程图自动布局（位于activiti-bpmn-layout模块）    #2
        new BpmnAutoLayout(model).execute();
        // 3. 把BpmnModel对象部署到引擎    #3
        Deployment deployment = activitiRule.getRepositoryService().createDeployment()
            .addBpmnModel("dynamic-model.bpmn", model)
            .name("Dynamic process deployment").deploy();
        // 4. 启动流程
        ProcessInstance processInstance = activitiRule.getRuntimeService()

```



```

        .startProcessInstanceByKey("my-process");
// 5. 检查流程是否正式启动
List<Task> tasks = activitiRule.getTaskService().createTaskQuery()
    .processInstanceId(processInstance.getId()).list();
// 验证Task
Assert.assertEquals(1, tasks.size());
Assert.assertEquals("First task", tasks.get(0).getName());
Assert.assertEquals("fred", tasks.get(0).getAssignee());
// 6. 导出流程图
InputStream processDiagram = activitiRule.getRepositoryService()
    .getProcessDiagram(processInstance.getProcessDefinitionId());
// 把文件生成在本章项目的test-classes目录中
String userHomeDir = getClass().getResource("/").getFile();
System.out.println(userHomeDir);
FileUtils.copyInputStreamToFile(processDiagram,
    new File(userHomeDir + "/diagram.png"));
// 7. 导出BPMN文件到本地文件系统
InputStream processBpmn = activitiRule.getRepositoryService()
    .getResourceAsStream(deployment.getId(), "dynamic-model.bpmn");
FileUtils.copyInputStreamToFile(processBpmn,
    new File(userHomeDir + "/process.bpmn20.xml"));
}
// 创建用户任务
protected UserTask createUserTask(String id, String name, String assignee) {
    UserTask userTask = new UserTask();
    userTask.setName(name);
    userTask.setId(id);
    userTask.setAssignee(assignee);
    return userTask;
}
protected SequenceFlow createSequenceFlow(String from, String to) { // 创建顺序流
    SequenceFlow flow = new SequenceFlow();
    flow.setSourceRef(from);
    flow.setTargetRef(to);
    return flow;
}
protected StartEvent createStartEvent() { // 创建启动事件
    StartEvent startEvent = new StartEvent();
    startEvent.setId("start");
    return startEvent;
}
protected EndEvent createEndEvent() { // 创建结束事件
    EndEvent endEvent = new EndEvent();
    endEvent.setId("end");
    return endEvent;
}
}
}

```

运行完代码清单21-3中的类后在本章示例的test-classes目录中可以看到生成两个文件：diagram.png、process.bpmn20，如图21-4所示。

在代码清单21-3中依次创建了开始事件、用户任务以及连接各个活动之间的顺序流。#2处的流程图自动布局需要特别说明，在#2处之前创建的对象没有任何坐标信息。我们知道要生成流程图，需要提供“bpmndi:BPMNDiagram”定义，也就是每一个元素的XY坐标以及宽度和高度信息。Activiti的**activiti-bpmn-layout**就是专门用来创建bpmndi信息的模块，该模块提供的**BpmnAutoLayout**类可以自动根据元素的类型计算出高度、宽度及XY坐标，然后设置到BpmnModel对象的“locationMap”属性中。图21-5展示了自动生成的流程图。

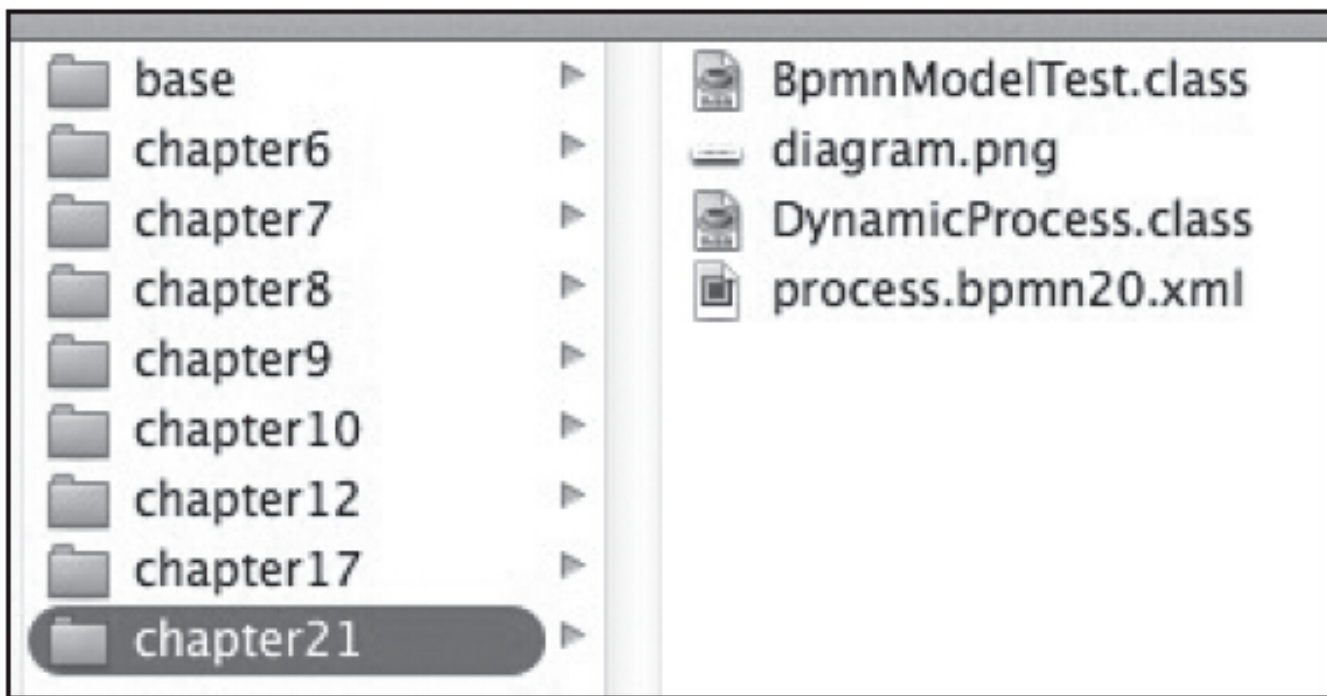


图21-4 由动态创建的流程生成的图片与BPMN文件

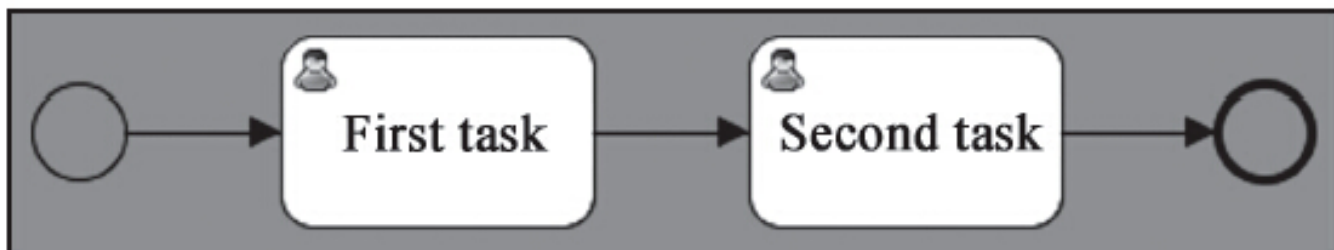


图21-5 动态创建流程自动布局后生成的流程图

介绍了这么多有关BpmnModel知识，可能会有人疑惑为什么要学习它？用它能做什么？

代码清单21-5的#1处返回了两种类型的BPMN 2.0元素对象，分别为Process（流程）、用户任务（UserTask），#2和#3处分别针对不同的类型进行了处理。运行代码清单21-6的测试后观察运行结果以理解其运作过程，当然建议读者用Debug的方式在Activiti源码中断点观察运行过程。

代码清单21-6 BpmnParseHandlerTest

```
public class BpmnParseHandlerTest extends AbstractTest {
    public BpmnParseHandlerTest() {
        super.activitiRule = new ActivitiRule("activiti.cfg.chapter21.parse.xml");
    }
    @Test
    @Deployment(resources = "chapter6/dynamic-form/leave.bpmn")
    public void testParseHandler () throws Exception {
        ProcessDefinition processDefinition = repositoryService
            .createProcessDefinitionQuery().singleResult();
        assertEquals("请假流程-动态表单-被PDE解析器修改", #1
            processDefinition.getName());
        assertEquals("leave-modified-by-post-parse-handler", processDefinition.getKey()); #2
    }
}
```

运行完单元测试后结果正确，#1和#2处的断言结果为true，说明代码清单21-4和代码清单21-5的解析处理器正常工作了，观察控制台还可以看到如下的输出（代码清单21-5中#3处循环语句输出的内容）：

```
UserTask:[部门领导审批]的输出流:
    exclusivegateway5
UserTask:[调整申请]的输出流:
    exclusivegateway7
UserTask:[人事审批]的输出流:
    exclusivegateway6
UserTask:[销假]的输出流:
    endevent1
```

除了以上两种处理方式外还可以覆盖引擎内部的解析器，例如把所有的User Task设置为异步，可以在引擎配置对象中设置属性customDefaultBpmnParseHandlers，相关配置如下，实现类见代码清单21-7。

```
<bean id="processEngineConfiguration" class="org.activiti
    .engine.impl.cfg.XxxProcessEngineConfiguration">
    <property name="customDefaultBpmnParseHandlers">
        <list>
            <bean class="me.kafeitu.activiti.chapter21
                .parse.CustomUserTaskBpmnParseHandler" />
        </list>
    </property>
</bean>
```

代码清单21-7 CustomServiceTaskBpmnParseHandler

```
public class CustomUserTaskBpmnParseHandler extends UserTaskParseHandler {
    protected void executeParse(BpmnParse bpmnParse, UserTask userTask) {
        super.executeParse(bpmnParse, userTask);
        ActivityImpl activity = findActivity(bpmnParse, userTask.getId());
        activity.setAsync(true);
    }
}
```

在代码清单21-6的基础上再添加一些输出（见代码清单21-8），查看控制台的输出结果，验证User Task是否被设置为异步。

 **提示** 自定义的解析处理器只会更改流程定义对象的属性，不会在数据库中更改BPMN文件内容。

代码清单21-8 用来验证User Task是否被更改成了异步

```
@Test
@Deployment(resources = "chapter6/dynamic-form/leave.bpmn")
public void testParseHandler() throws Exception {
    ProcessDefinition processDefinition = repositoryService
        .createProcessDefinitionQuery().singleResult();
    RepositoryServiceImpl repositoryServiceImpl = (RepositoryServiceImpl) repositoryService;
    ReadOnlyProcessDefinition deployedProcessDefinition = repositoryServiceImpl
        .getDeployedProcessDefinition(processDefinition.getId());
    ProcessDefinitionEntity processDefinitionEntity = (ProcessDefinitionEntity) deployedProcessDefinition;
    List<ActivityImpl> activitiList = processDefinitionEntity.getActivities();
    for (ActivityImpl activity : activitiList) {
        System.out.println("Activity Name: " + activity.getProperty("name")
            + ", async=" + activity.isAsync());
    }
}
```

执行完代码清单21-8后输出的结果如下，从结果中可以看出所有的用户任务类型的异步属性均为true。

```
Activity Name: 部门领导审批, async=true
Activity Name: Exclusive Gateway, async=false
Activity Name: 调整申请, async=true
Activity Name: 人事审批, async=true
Activity Name: Exclusive Gateway, async=false
Activity Name: 销假, async=true
Activity Name: Exclusive Gateway, async=false
Activity Name: Start, async=false
Activity Name: End, async=false
```

[1] <http://stacktrace.be/blog/2013/03/dynamic-process-creation-and-deployment-in-100-lines>

21.2 全局事件处理器

我们知道流程（Process）可以定义start、end事件来监听流程的开始与结束，也可以在用户任务上定义任务的create、assignment、complete、delete事件，可以在输出流（flow）上定义take事件。这些事件监听器被预先分散地定义在每一个流程文件中，如果某个流程在运行中需要添加一个监听器，只能选择重新设计流程，然后部署。另外很多功能重复的监听器被定义在不同的流程文件中也不便于日后的维护。

有没有办法解决这些问题呢？很庆幸，从Activiti 5.15版本开始添加了全局事件处理器（Event handler）。在官方提供的用户手册中没有出现Global Event Handler，但是笔者认为应该翻译为全局事件处理器，因为这些事件处理器可以作用于一个引擎实例的所有流程或者作用于一个流程实例，而不是把事件绑定在某一个活动（Activity）上。

全局事件处理器这种事件处理方式的工作原理是在某一个事件被触发时由引擎发送通知（Notify），根据事件的名称匹配所有注册到引擎的事件处理器，这些处理器都会收到事件通知（典型的监听者模式）：可以在引擎配置对象中配置事件处理器，也可以在流程文件中配置（只能在Process中配置，不能在某一个Activity上配置），也可以在运行时通过引擎提供的API添加事件处理器（不过运行时添加的事件处理器仅存在于内存中，重启引擎后就会被销毁）。表21-1列出了引擎内置的所有事件名称及事件类型（类名为ActivitiXxxEvent）。

表21-1 引擎内置的事件名称及事件类型

事件名称	描述	事件类型
ENGINE_CREATED	监听器监听的流程引擎已经创建完毕，并且准备好接受 API 调用	ActivitiEvent
ENGINE_CLOSED	监听器监听的流程引擎已经关闭，不再接受 API 调用	ActivitiEvent
ENTITY_CREATED	创建了一个新实体，可以从事件中获取实体	ActivitiEntityEvent
ENTITY_INITIALIZED	创建了一个新实体，初始化也完成了。如果这个实体的创建会包含子实体的创建，这个事件会在子实体都创建 / 初始化完成后被触发，这是与 ENTITY_CREATED 的区别	ActivitiEntityEvent
ENTITY_UPDATED	更新了已存在的实体，可以从事件中获取实体	ActivitiEntityEvent
ENTITY_DELETED	删除了已存在的实体，可以从事件中获取实体	ActivitiEntityEvent
ENTITY_SUSPENDED	暂停了已存在的实体，可以从事件中获取实体。会被 ProcessDefinitions, ProcessInstances 和 Tasks 抛出	ActivitiEntityEvent
ENTITY_ACTIVATED	激活了已存在的实体，可以从事件中获取实体。会被 ProcessDefinitions, ProcessInstances 和 Tasks 抛出	ActivitiEntityEvent
JOB_EXECUTION_SUCCESS	作业执行成功，Job 对象包含在事件中	ActivitiEntityEvent
JOB_EXECUTION_FAILURE	作业执行失败。作业和异常信息包含在事件中	ActivitiEntityEvent ActivitiExceptionEvent
JOB_RETRIES_DECREMENTED	因为作业执行失败，导致重试次数减少。作业包含在事件中	ActivitiEntityEvent
TIMER FIRED	触发了定时器，Job 对象包含在事件中	ActivitiEntityEvent
ACTIVITY_STARTED	一个节点开始执行	ActivitiActivityEvent
ACTIVITY_COMPLETED	一个节点成功结束	ActivitiActivityEvent
ACTIVITY_TIMEOUT	一个活动的定时边界事件超时	ActivitiActivityEvent
ACTIVITY_SINGALED	一个节点收到了一个信号	ActivitiSignalEvent

(续)

事件名称	描述	事件类型
ACTIVITY_MESSAGE_RECEIVED	一个节点收到了一个消息。在节点收到消息之前触发，而在收到后，会触发 ACTIVITY_SIGNAL 或 ACTIVITY_STARTED，根据节点的类型（边界事件，事件子流程开始事件）不同触发的事件也不同	ActivitiMessageEvent
ACTIVITY_ERROR_RECEIVED	一个节点收到了一个错误事件。在节点实际处理错误之前触发。事件的 activityId 对应着处理错误的节点。这个错误发送成功，后续会触发 ACTIVITY_SIGNALLED 或 ACTIVITY_COMPLETE 其中一种	ActivitiErrorEvent
UNCAUGHT_BPMN_ERROR	抛出了未捕获的 BPMN 错误。流程没有提供针对这个错误的处理器。事件的 activityId 为空	ActivitiErrorEvent
ACTIVITY_COMPENSATE	一个节点将要被补偿。事件包含了将要执行补偿的节点 id	ActivitiActivityEvent
VARIABLE_CREATED	创建了一个变量。事件包含变量名、变量值和对应的分支或任务（如果存在）	ActivitiVariableEvent
VARIABLE_UPDATED	更新了一个变量。事件包含变量名、变量值和对应的分支或任务（如果存在）	ActivitiVariableEvent
VARIABLE_DELETED	删除了一个变量。事件包含变量名、变量值和对应的分支或任务（如果存在）	ActivitiVariableEvent
TASK_CREATED	创建了任务实体并且任务的所有属性都已设置	ActivitiEntityEvent
TASK_ASSIGNED	任务被分配给了一个人员。事件包含任务	ActivitiEntityEvent
TASK_COMPLETED	任务被完成了。它会在 ENTITY_DELETE 事件之前触发。当任务是流程一部分时，事件会在流程继续运行之前，后续事件将是 ACTIVITY_COMPLETE，对应着完成任务的节点	ActivitiEntityEvent
PROCESS_COMPLETED	流程已结束，当流程的最后一个活动触发了 ACTIVITY_COMPLETED 事件后该事件被触发	ActivitiEntityEvent
MEMBERSHIP_CREATED	用户被添加到一个组里。事件包含了用户和组的 id	ActivitiMembershipEvent
MEMBERSHIP_DELETED	用户被从一个组中删除。事件包含了用户和组的 id	ActivitiMembershipEvent
MEMBERSHIPS_DELETED	所有成员被从一个组中删除。在成员删除之前触发这个事件，所以它们都是可以访问的。从性能方面考虑，不会为每个成员触发单独的 MEMBERSHIP_DELETED 事件	ActivitiMembershipEvent

21.2.1 定义事件处理器

预先定义事件处理器的方式有两种：在引擎配置文件中定义和在流程文件中定义。事件处理器均需集成自接口 `org.activiti.engine.delegate.event.ActivitiEventListener`。

1.在引擎配置文件中定义

在引擎配置对象中配置属性“eventListeners”，下面的代码列出了相关配置。而事件处理器类 `GlobalEventListener` 的实现见代码清单21-9所示，简单输出所有的事件名称。

```
<bean id="processEngineConfiguration"
  class="org.activiti.XxxEngineConfiguration">
  <property name="eventListeners">
    <list>
      <bean class="me.kafeitu.activiti.chapter21
        .events.GlobalEventListener" />
    </list>
  </property>
</bean>
```

代码清单21-9 事件处理器GlobalEventListener

```
public class GlobalEventListener implements ActivitiEventListener {
    @Override
    public void onEvent(ActivitiEvent event) {
        System.out.println("捕获到事件: " + event.getType().name() + ", type=" +
            ToStringBuilder.reflectionToString(event));
    }
    @Override
    public boolean isFailOnException() {
        return false;
    }
}
```

代码清单21-10中的测试类运行一个完整的流程，测试代码来源于第6章的动态表单测试代码，不过由于篇幅有限不再重复介绍测试用例。

代码清单21-10 GlobalEventHandlerTest

```
public class GlobalEventHandlerTest extends AbstractTest {
    @Test
    public void testEventListener() throws Exception {
        initProcessEngine("chapter21/activiti.cfg.chapter21.event.eventListener.xml");
        deploy("chapter21/leave.bpmn");
        execute(); // 参考LeaveDynamicFormTest类的allApproved方法
    }
}
```

在执行完代码清单21-10的测试用例后观察控制台，输出的内容如下，从中可以看出每捕获到一个事件后控制台输出。

```
捕获到事件: ENGINE_CREATED, type=org.activiti.engine.delegate.event.impl.ActivitiEventImpl@520b1684[type=ENGINE_CREATED,executionId=<null>,processInstanceId=<null>,processDefinitionId=<null>]
捕获到事件: ENTITY_CREATED, type=org.activiti.engine.delegate.event.impl.ActivitiEntityEventImpl@72a60191[entity=DeploymentEntity[id=1, name=GlobalEventHandlerTest.allApproved],type=ENTITY_CREATED,executionId=<null>]
捕获到事件: ENTITY_INITIALIZED, type=org.activiti.engine.delegate.event.impl.ActivitiEntityEventImpl@525dc268[entity=ProcessDefinitionEntity[leave:1:3],type=ENTITY_CREATED,executionId=<null>]
捕获到事件: ENTITY_INITIALIZED, type=org.activiti.engine.delegate.event.impl.ActivitiEntityEventImpl@75e6743e[entity=ProcessDefinitionEntity[leave:1:3],type=ENTITY_INITIALIZED,executionId=<null>]
捕获到事件: ENTITY_INITIALIZED, type=org.activiti.engine.delegate.event.impl.ActivitiEntityEventImpl@4f9f9c7a[entity=DeploymentEntity[id=1, name=GlobalEventHandlerTest.allApproved],type=ENTITY_INITIALIZED,executionId=<null>]
捕获到事件: VARIABLE_CREATED, type=org.activiti.engine.delegate.event.impl.ActivitiVariableEventImpl@705ba50e[variableName=applyUserId,variableValue=henryyan,taskId=<null>,type=VARIABLE_CREATED,executionId=<null>]
...
```

属性“eventListeners”定义的事件处理器覆盖了全部类型的事件（见表21-1），当然也可以只处理指定类型的事件，把属性“eventListeners”更换成“typedEventListeners”定义事件类型与事件处理器的映射关系。下面代码中配置了两个事件类型，每个事件类型用逗号分割。

```
<bean id="processEngineConfiguration"
      class="org.activiti.XxxEngineConfiguration">
    <property name="typedEventListeners">
        <map>
            <entry key="ENGINE_CREATED,ENTITY_CREATED">
                <list>
                    <bean class="me.kafeitu.activiti
                        .chapter21.events.GlobalEventListener" />
                </list>
            </entry>
        </map>
    </property>
</bean>
```

执行下面的测试方法，然后从结果中过滤，可以看到只有定义的两类型的事件被触发。

```
@Test
public void testTypedEventListener() throws Exception {
    initProcessEngine("chapter21/activiti.cfg
        .chapter21.event.typedEventListener.xml");
    deploy("chapter21/leave.bpmn");
    execute();
}
```

2.在流程文件中定义

在引擎配置文件中定义的事件处理器作用域是整个引擎，也就是说引擎中所有流程实例触发的事件均会被定义的事件处理器捕获，可能有些事件只作用于某一个流程，所以引擎也支持在流程中定义事件处理器。下面的配置在流程文件的BPMN 2.0扩展标签extensionElement中添加一个“activiti:eventListener”标签，用来定义和当前流程有关的事件处理器。

```
<process id="leave" name="请假流程">
    <extensionElements>
        <activiti:eventListener class="me.kafeitu.activiti
            .chapter21.events.GlobalEventListener" />
    </extensionElements>
    http://www.hzcourse.com/resource/readBook?path=/openresources/teach_ebook/uncompressed/15030/OEBPS/Text/...
</process>
```

执行下面的单元测试后分析控制台的输出结果，可以看到与代码清单21-10的输出结果一致。

```
@Test
public void testEventInProcess() throws Exception {
    initProcessEngine(
        "chapter21/activiti.cfg.chapter21.event.default.xml");
    deploy("chapter21/leave-event.bpmn");
    execute();
}
```

在流程文件中除了定义一个事件处理器类之外，还可以把捕获的事件转换为消息事件、信号事件、错误事件等。下面的几个示例配置定义了事件并触发其他事件。

以下配置表示：当触发TASK_CREATED事件时，抛出名称为“SIGNAL_001”的信号事件，当然也可以抛出一个全局类型的事件（throwEvent=“globalSignal”）。

```
<process id="testEventListeners">
    <extensionElements>
        <activiti:eventListener throwEvent="signal"
            signalName="SIGNAL_001" events="TASK_CREATED" />
    </extensionElements>
</process>
```

以下配置表示：当触发TASK_CREATED事件时抛出一个名称为“MSG_001”的事件，通过这个信号事件可以启动一个流程或触发中间捕获事件。

```
<process id="testEventListeners">
    <extensionElements>
        <activiti:eventListener throwEvent="message"
            messageName="MSG_001" events="TASK_CREATED" />
    </extensionElements>
</process>
<process id="testEventListeners">
    <extensionElements>
        <activiti:eventListener throwEvent="error"
            errorCode="ERR_001" events="TASK_CREATED" />
    </extensionElements>
</process>
```

 说明 全局事件在当前Activiti版本（5.15）中还是实验性的，后续版本可能会有改动。

21.2.2 处理捕获的事件

上一小节只是简单介绍了如何定义事件处理器，自定义的事件处理器类GlobalEvent-Listener也只是对事件的类型做了输出而已，本小节将详细介绍处理不同类型的事件时如何区分事件类型，以及如何获取相关数据。

在代码清单21-9中，接口ActivitiEventListener的onEvent方法的ActivitiEvent参数定义了事件对象可以获取的一些参数，例如获取当前事件的事件类型、流程实例ID、流程定义ID，也可以获取EngineServices接口从而获取引擎的7个Service接口。

为了更清楚地了解不同事件的处理方式，将代码清单21-9的示例加以改造，需要针对不同的事件做一些翻译工作。不同类型的事件在处理时需要将类型强制转换成具体的事件对象，例如，在代码清单21-11中处理了ENTITY_CREATED事件，在控制台打印是哪个实体被创建。

代码清单21-11 在事件处理器中区分不同事件类型

```
public class GlobalEventListener implements ActivitiEventListener {
    @Override
    public void onEvent(ActivitiEvent event) {
        ActivitiEventType eventType = event.getType(); // 获取事件类型（枚举）
        switch (eventType) {
            case ENGINE_CREATED:
                System.out.println("引擎初始化成功！");
                break;
            case ENGINE_CLOSED:
                System.out.println("引擎已关闭！");
                break;
            case ENTITY_CREATED:
                // 类型强制转换为实际类型，参考表21-1
                ActivitiEntityEvent entityEvent = (ActivitiEntityEvent) event;
                System.out.println("创建了实体: " + entityEvent.getEntity());
                break;
            case ENTITY_INITIALIZED:
                entityEvent = (ActivitiEntityEvent) event;
                System.out.println("实体初始化完毕: " + entityEvent.getEntity());
                break;
            default:
                System.out.println("捕获到事件[需要处理]: " + eventType.name() + ",
                    type=" + ToStringBuilder.reflectionToString(event));
        }
    }
    @Override
    public boolean isFailOnException() { return false; }
}
```

可以参照代码清单21-11中的代码以及表21-1的事件类型与事件类的对照表，添加不同事件的处理。

事件处理器除了实现ActivitiEventListener接口外还可以继承引擎封装的基类，如专门用来处理实体相关事件的处理基类BaseEntityEventListener，该类实现ActivitiEventListener接口并过滤实体(Entity)类型的事件，封装了onCreate()、onInitialized()、onDelete()、onUpdate()方法来处理不同类型的事件，在使用时继承该类并覆盖需要的方法即可。代码清单21-12列出了继承自BaseEntityEventListener的事件处理器类EntityEvent-Listener。

代码清单21-12 专门用来处理实体类型事件的事件处理器

```
public class EntityEventListener extends BaseEntityEventListener {
    @Override
    protected void onCreate(ActivitiEvent event) {
        ActivitiEntityEvent entityEvent = (ActivitiEntityEvent) event;
        System.out.println("创建了实体: " + entityEvent);
    }
    @Override
    protected void onInitialized(ActivitiEvent event) {
        super.onInitialized(event);
    }
    @Override
    protected void onDelete(ActivitiEvent event) {
        ActivitiEntityEvent entityEvent = (ActivitiEntityEvent) event;
        System.out.println("删除了实体: " + entityEvent);
    }
    @Override
    protected void onUpdate(ActivitiEvent event) {
        super.onUpdate(event);
    }
    @Override
    protected void onEntityEvent(ActivitiEvent event) {
        super.onEntityEvent(event);
    }
}
```

代码清单21-12的测试方法如下，读者可以实际操作一下，然后观察控制台打印的结果。

```
@Test
public void testEntityEvent() throws Exception {
    initProcessEngine(
        "chapter21/activiti.cfg.chapter21.event.entity.xml");
    deploy("chapter21/leave.bpmn");
    execute();
}
```

21.2.3 事件处理器的异常处理

事件处理器接口ActivitiEventListener中的isFailOnException()方法的作用还未提及，该方法需要返回一个布尔类型的值，其返回值决定了当方法onEvent()抛出异常时引擎如何处理异常：当返回的值为false时忽略异常程序继续执行，当返回值为true时引擎会把onEvent()抛出的异常继续向上传播。所以一般会在事件处理器中定义一个成员变量，在方法onEvent()中根据不同条件决定该变量的true与false。代码清单21-13展示了一个处理异常的示例。

代码清单21-13 事件处理器的异常处理

```
public class EventExceptionListener implements ActivitiEventListener {
    private boolean isFailOnException = false;
    @Override
    public void onEvent(ActivitiEvent event) {
        ActivitiEventType eventType = event.getType();
        switch (eventType) {
            case ENTITY_CREATED:
                ActivitiEntityEvent entityEvent = (ActivitiEntityEvent) event;
                System.out.println("创建了实体: " + entityEvent.getEntity());
                break;
            case ENTITY_DELETED:
                entityEvent = (ActivitiEntityEvent) event;
                if (entityEvent.getEntity() instanceof TaskEntity) {
                    isFailOnException = true;
                }
                break;
        }
    }
}
```



```
        throw new RuntimeException("不允许删除TaskEntity");
    }
    System.out.println("实体已被删除: " + entityEvent.getEntity().getClass());
    break;
}
}
@Override
public boolean isFailOnException() {
    return isFailOnException;
}
}
```

代码清单21-13中加粗部分判断了实体的类型，如果是TaskEntity类型，在删除时抛出异常并设置了变量isFailOnException为true，这样当抛出RuntimeException异常时引擎不会忽略异常而是继续向上传播，最终导致操作失败，产生事务回滚。执行下面的测试后单元测试执行失败，控制台也打印了异常信息。

```
@Test
public void testEventException() throws Exception {
    initProcessEngine(
        "chapter21/activiti.cfg.chapter21.event.exception.xml");
    deploy("chapter21/leave.bpmn");
    execute();
}
```

控制台输出的异常信息：

```
Caused by: java.lang.RuntimeException: 不允许删除TaskEntity
    at me.kafetu.activiti.chapter21.events.EventExceptionHandler.onEvent(EventExceptionHandler.java:31)
    at org.activiti.engine.delegate.event.impl.ActivitiEventSupport.dispatchEvent(ActivitiEventSupport.java:104)
    http://www.hzcourse.com/resource/readBook?path=/openresources/teach_ebook/uncompressed/15030/OEBPS/Text/... 40 more
```

21.2.4 动态注册事件处理器

在本节（21.2节）开头提到了可以动态注册、删除事件处理器，值得注意的是在引擎销毁或重新启动后动态注册的事件处理器将被销毁，要了解原因首先要分析一下事件处理器的管理机制。

在初始化引擎时会创建一个空的事件调度器（Event Dispatcher），事件调度器在内部会维护两个空的集合（称之为事件处理器池）：一个是处理所有类型的事件处理器（List<ActivitiEventListener>），另外一个区分事件类型的Map集合（Map<ActivitiEventType, List<ActivitiEventListener>>）。

事件调度器在初始化完成后会读取引擎配置文件中的事件处理器集合，然后加入到事件调度器中。在读取一个流程定义时需要解析流程文件，如果流程文件中定义了事件处理器池，从流程文件中获取的事件处理器也会加入到事件调度器维护的事件处理器池中。

为了动态维护事件处理器池引擎开放了相应的接口，在RuntimeService接口中有以下几个方法可供使用。

```
· void addEventListener(ActivitiEventListener listenerToAdd);

· void addEventListener(ActivitiEventListener listenerToAdd, ActivitiEventTypehttp://www.hzcourse.com/resource/readBook?path=/openresources/teach_ebook/uncompressed/15030/OEBPS/Text/...types);

· void removeEventListener(ActivitiEventListener listenerToRemove);
```

可以使用addEventListener方法增加一个事件处理器（处理所有事件类型），也可以通过types参数限定该事件处理器支持哪些事件类型，当然也可以调用方法removeEventListener移除一个事件处理器。

21.2.5 任务自动转办

想象一下这样的场景：员工张三每天需要在系统中处理大约10项待办任务，因为公司的业务发展需要派遣张三到外地出差一周，而原本的日常工作需要在公司的内部系统中处理（并且张三负责的任务在整个流程中比重相对较高）。

张三在外地如何处理日常的工作呢？可以考虑使用VPN技术让他登录到公司内部系统，也可以把任务转办给其他的同事代办。本节内容就来讲解一下如何实现任务的自动转办功能。

自动转办指分配给某个员工的任务自动转交给预设（任务的原办理人）的转办人。当然转办可以配置一些规则或者限制，例如可以设置哪些流程、哪些任务，或者限制在一段时间之内待办任务项由系统自动转办给转办人代办。

相对于自动转办的文字描述来说，在Activiti中用代码实现要简单得多，最重要的一个动作就是更改任务的办理人（assignee）属性，当然在更改办理人属性之前会针对一些条件做出判断处理，只有在满足预设的条件之后才能真正把任务转办给其他人处理。

任务自动转办的实现方式有两种，一种是利用引擎提供的任务监听器（UserTask的create事件），另外一种就是利用全局事件处理器（TASK_CREATED事件类型）。下面分别介绍两种方式的实现方法。

1.使用任务监听器

在本章源码src/test/resources/chapter21/chapter21/leave-auto-delegate.bpmn流程文件中，在“销假”节点处添加“create”类型的任务监听器，相关配置如下：

```
<userTask id="reportBack" name="销假"
    activiti:assignee="${applyUserId}">
    <extensionElements>
        <activiti:taskListener event="create"
            class="me.kafetu.activiti.chapter21.listeners.TaskAutoRedirectListener" />
    </extensionElements>
</userTask>
```

TaskAutoRedirectListener的实现代码如代码清单21-14所示，在代码中用一个静态的Map对象预先定义任务办理人的转办关系，在实际中可以从用户配置（保存在数据库）中读取。

代码清单21-14 TaskAutoRedirectListener

```
public class TaskAutoRedirectListener implements TaskListener {
    private static Map<String, String> userMap = new HashMap<String, String>();
    static { userMap.put("henryan", "thomas"); }
    @Override
    public void notify(DelegateTask delegateTask) {
        String originAssignee = delegateTask.getAssignee(); // 原任务办理人
        String newUser = userMap.get(originAssignee); // 预设的转办人
        if (StringUtil.isNotBlank(newUser)) {
```

```
        delegateTask.setAssignee(newUser); // 把任务重新分配给转办人
        EngineServices engineServices =
            delegateTask.getExecution().getEngineServices();
        TaskService taskService = engineServices.getTaskService();
        String message = getClass().getName() + "> 任务[" + delegateTask.
            taskService.addComment(delegateTask.getId(), // 记录转办事件
                delegateTask.getProcessInstanceId(), "redirect", message);
    } else {
        System.out.println("任务[" + delegateTask.getName() + "]没有预设的转办人");
    }
}
}
```

简单分析一下代码清单21-14的监听器处理逻辑：如果请假流程由用户henryyan发起，并且henryyan用户预先设置了任务自动转办给用户thomas，结果销假节点的办理人应该为thomas而不是henryyan。

更改GlobalEventHandlerTest类的execute()方法中的销假任务办理人断言，用来验证任务的办理人为henryyan，以此判断代码清单21-14的监听器是否执行成功。

```
String currentUserId = "henryyan";
Task reportBackTask =
    taskService.createTaskQuery().singleResult();
List<Comment> taskComments = taskService
    .getTaskComments(reportBackTask.getId(), "delegate");
for (Comment taskComment : taskComments) {
    System.out.println("任务相关事件: " +
        taskComment.getFullMessage());
}
// 判断
assertEquals(currentUserId, reportBackTask.getAssignee());
```

执行下面的单元测试验证执行结果：

```
@Test
public void testAutoRedirectUseTaskListener() {
    initProcessEngine(
        "chapter21/activiti.cfg.chapter21.event.default.xml");
    deploy("chapter21/leave-auto-redirect.bpmn");
    execute();
}
```

单元测试的执行结果为失败，观察控制台可以看到如下的输出结果：

```
任务相关事件:
me.kafeitu.activiti.chapter21.listeners.TaskAutoRedirectListener-> 任务[销假]的办理人[henryyan]自动转办给了用户[thomas]
org.junit.ComparisonFailure:
Expected :henryyan
Actual   :thomas
```

2.使用全局事件处理器

相对于使用监听器实现任务自动转办来说，使用全局事件处理器来实现更为方便，因为使用监听器方式实现必须为每个流程的用户任务添加“create”监听器，当然也可以使用21.1节介绍的BPMN解析处理器拦截UserTask，然后动态添加监听器实现。

使用全局事件处理器实现任务自动转办需要定义TASK_CREATED事件处理器，该事件处理器和任务的“create”监听器类似，也就是在任务对象初始化并设置好所有属性后被触发。

需要提醒读者的是TASK_CREATED事件从Activiti 5.16版本才开始支持。下面的配置代码定义了一个TASK_CREATE类型的全局事件处理器。

```
<property name="typedEventListeners">
    <map>
        <entry key="TASK_CREATED"> <!-- 5.16版本开始支持 -->
            <list>
                <!-- 任务自动转办 -->
                <bean class="me.kafeitu.activiti.chapter21.
                    events.TaskAutoRedirectGlobalEventListener" />
            </list>
        </entry>
    </map>
</property>
```

代码清单21-15的处理逻辑与代码清单21-14类似，先预先设置了一个代办对应关系变量userMap，再判断实体类型是否为TaskEntity，最后判断任务的办理人是否设置了代理人。

代码清单21-15 TaskAutoRedirectGlobalEventListener

```
public class TaskAutoRedirectGlobalEventListener implements ActivitiEventListener {
    private static Map<String, String> userMap = new HashMap<String, String>();
    static {
        userMap.put("henryyan", "thomas");
    }
    @Override
    public void onEvent(ActivitiEvent event) {
        ActivitiEntityEvent entityEvent = (ActivitiEntityEvent) event;
        Object entity = entityEvent.getEntity();
        if (entity instanceof TaskEntity) { // 用户任务类型的实体
            TaskEntity task = (TaskEntity) entity;
            String originUserId = task.getAssignee();
            String newUserId = userMap.get(originUserId);
            if (StringUtils.isNotBlank(newUserId)) {
                task.setAssignee(newUserId); // 重新设置任务办理人
            }
        }
    }
    @Override
    public boolean isFailOnException() { return false; }
}
```

执行下面的单元测试方法观察结果，与代码清单21-14的输出结果一致：

```
@Test
public void testAutoRedirectUseGlobalEvent() {
    initProcessEngine(
        "chapter21/activiti.cfg.chapter21.event.auto-redirect.xml");
    deploy("chapter21/leave.bpmn");
    execute();
}
```

在执行上面的单元测试后，JUnit提示断言失败，控制台输出异常信息：

```
org.junit.ComparisonFailure:
```

21.2.6 事件日志

Activiti 5.16版本新增了全局事件的日志表（ACT_EVT_LOG），该功能默认未启用，要启用可以通过更改引擎配置属性来实现，开启全局事件日志后当引擎触发全局事件后会记录到表ACT_EVT_LOG中。值得注意的是并不是所有的事件都会被记录，引擎预设了以下几大类事件类型。

任务相关事件类型：

- TASK_CREATED，任务创建。
- TASK_COMPLETED，任务完成。
- TASK_ASSIGNED，任务分配办理人。

输出流相关事件类型：SEQUENCEFLOW_TAKEN，执行Flow时。

活动（Activity）相关事件类型：

- ACTIVITY_COMPLETED，活动完成后。
- ACTIVITY_STARTED，活动开始执行。
- ACTIVITY_SIGNED，活动接收到信号。
- ACTIVITY_MESSAGE_RECEIVED，活动接收到消息。
- ACTIVITY_COMPENSATE，活动执行了补偿后。
- ACTIVITY_ERROR_RECEIVED，活动接收到错误信号。

变量相关事件类型：

- VARIABLE_CREATED，变量对象创建后。
- VARIABLE_DELETED，变量被删除后。
- VARIABLE_UPDATED，变量被更新后。

开启全局事件日志功能可以设置引擎的“enableDatabaseEventLogging”参数为true，相关配置如下：

```
<bean id="processEngineConfiguration"
      class="org.activiti.xxx.XxxProcessEngineConfiguration">
    <property name="enableDatabaseEventLogging"
              value="true" />
</bean>
```

在本章示例中启动一个流程，然后再观察ACT_EVT_LOG表就可以看到各种类型的日志记录，如图21-6所示。

从图21-6中可以看到包含了一系列和流程数据有关的日志，“TYPE_”字段为事件类型，“DATA_”字段为日志内容，但该字段的值为二进制类型（由JSON格式的字符串序列化），在通过下面的API读取到日志数据后需要把二进制转换为String对象。

```
ManagementService managementService = ...
managementService.getEventLogEntries(start, pageSize);
```

SELECT * FROM ACT_EVT_LOG;								
LOG_NR_	TYPE_	PROC_DEF_ID_	PROC_INST_ID_	EXECUTION_ID_	TASK_ID_	TIME_STAMP_	USER_ID_	DATA_
1	VARIABLE_CREATED	leave:1:31	102	102	null	2014-07-26 22:10:08.997	kermit	7b227661726961626c65547
2	PROCESSINSTANCE_START	leave:1:31	102	null	null	2014-07-26 22:10:09.006	kermit	7b2274656e616e744964223
3	ACTIVITY_STARTED	leave:1:31	102	102	null	2014-07-26 22:10:09.006	kermit	7b226163746976697479496
4	ACTIVITY_COMPLETED	leave:1:31	102	102	null	2014-07-26 22:10:09.007	kermit	7b226163746976697479496
5	SEQUENCEFLOW_TAKEN	leave:1:31	102	102	null	2014-07-26 22:10:09.007	kermit	7b226964223a22666c6f7732
6	ACTIVITY_STARTED	leave:1:31	102	102	null	2014-07-26 22:10:09.008	kermit	7b226163746976697479496
7	TASK_CREATED	leave:1:31	102	102	107	2014-07-26 22:10:09.011	kermit	7b2274656e616e744964223

图21-6 ACT_EVT_LOG表的数据

当然也可以根据流程实例ID读取日志列表，代码如下：

```
managementService.
getEventLogEntriesByProcessInstanceId (processInstanceId);
```

以上两个接口均返回一个EventLogEntry类型的List集合，在展示日志内容时可以使用下面的代码把EventLogEntry的getData()方法返回的字节数组转换为字符串。

```
new String(logEntry.getData(), "UTF-8")
```

转换后的内容以为JSON字符串，如下的JSON内容为PROCESSINSTANCE_START事件记录。

```
{
  "tenantId": "",
  "id": "102",
  "createTime": 1406383809006,
  "timeStamp": 1406383809006,
  "userId": "kermit",
  "businessKey": "1",
  "processDefinitionId": "leave:1:31",
  "variables": {
    "applyUserId": "kermit"
  }
}
```

在本章示例的“管理”菜单添加了“事件日志”菜单项，页面展示的效果如图21-7所示。

单击图21-7中的“查看”按钮可以查看JSON数据，在遍历JSON数据后逐行输出到如图21-8的对话框中展示。

有关代码读者可以查看本章示例代码的EventLogController类及event-log.jsp文件。

日志ID	流程实例ID	执行ID	流程定义ID	任务ID	发生时间	类型	操作人	日志内容
event-log-1	102	102	leave:1:31		Sat Jul 26 22:10:08 CST 2014	VARIABLE_CREATED	kermit	查看
event-log-2	102		leave:1:31		Sat Jul 26 22:10:09 CST 2014	PROCESSINSTANCE_START	kermit	查看
event-log-3	102	102	leave:1:31		Sat Jul 26 22:10:09 CST 2014	ACTIVITY_STARTED	kermit	查看
event-log-4	102	102	leave:1:31		Sat Jul 26 22:10:09 CST 2014	ACTIVITY_COMPLETED	kermit	查看
event-log-5	102	102	leave:1:31		Sat Jul 26 22:10:09 CST 2014	SEQUENCEFLOW_TAKEN	kermit	查看

共 14 条 << < 1 2 3 > >>

图21-7 事件日志列表

查看日志详细

[← 上一条](#)[下一条 →](#)

tenantId

id

createTime

timeStamp

userId

businessKey

processDefinitionId

variables

logId

102

1406383809006

1406383809006

kermit

1

leave:1:31

applyUserId: kermit

event-log-2

图21-8 查看日志详细

21.3 命令与拦截器

外观模式（Facede）、命令模式（Command）、拦截器（Interceptor）模式是常见的设计模式，也是Activiti整体架构采用的三种主要设计模式，所有的API均以这三种模式为基础实现。

“面向接口编程”是Activiti API设计的一大设计思想，对外公开的所有API均用抽象定义的方式，提供了7大模块的XxxService接口，这也是采用了外观模式的表现；引擎的内部实现不对外公开，开发人员只要调用暴露的API编程接口实现响应的功能。基于抽象的接口还可以覆盖引擎内部的实现，使用自定义的实现类替换（在引擎配置对象中可以配置）。

在第1章介绍Activiti特性时就提到了命令模式，在Activiti中所有的操作均对应一个命令接口（Command）实现类，这样不同的功能分散在N个命令中，易于维护与扩展；当调用引擎XxxService接口时实际上是调用不同的Command，但接口并不是直接调用命令，而是把命令交给CommandExecutor统一执行，因为通过它可以在命令执行过程中执行若干拦截器（类似JAVAE中的Filter链）。

拦截器的作用顾名思义，Spring中的AOP就是一个典型的应用，可以在方法调用之前或之后执行拦截器，从而实现事务管理。在Activiti中拦截器也发挥了类似的作用，通过拦截器可以为所有的Command准备好命令上下文（CommandContext）对象，以便在Command实现类中获取到引擎配置对象（获取引擎配置属性）、会话对象（Session），以及其他扩展属性等；还可以为Command提供事务管理器、乐观锁自动重试等功能。

21.3.1 命令与拦截器运行机制

所有的命令均需实现org.activiti.engine.impl.interceptor.Command接口，该接口定义见代码如下：

```
public interface Command<T> {  
    T execute(CommandContext commandContext);  
}
```

 **说明** 在学习本节内容时建议用IDE打开Activiti源码中的ProcessEngineConfiguration-Impl类查看。

该接口仅定义了一个execute方法，其中最关键的是该方法的CommandContext参数，该参数为所有的命令提供了获取数据库、事务管理器、扩展属性等资源。

引擎在初始化时会初始化一系列命令拦截器，其中一个最重要的拦截器为CommandContextInterceptor，接口Command的CommandContext参数就是由它来提供。要分析拦截器如何为Command提供CommandContext参数，需要先了解一下Activiti中拦截器的运行过程。

下面先分析一下引擎源码中的一段有关默认拦截器的代码片段：

```
protected Collection<? extends CommandInterceptor>  
getDefaultCommandInterceptors() {  
    List<CommandInterceptor> interceptors =  
        new ArrayList<CommandInterceptor>();  
    interceptors.add(new LogInterceptor()); // 日志拦截器  
    // 事务拦截器  
    CommandInterceptor transactionInterceptor =  
        createTransactionInterceptor();  
    if (transactionInterceptor != null) {  
        interceptors.add(transactionInterceptor);  
    }  
    interceptors.add( // 命令上下文拦截器  
        new CommandContextInterceptor(  
            commandContextFactory, this));  
    return interceptors;  
}
```

再来分析读取所有拦截器的initCommandInterceptors方法()：

```
protected void initCommandInterceptors() {  
    if (commandInterceptors==null) {  
        commandInterceptors = new ArrayList<CommandInterceptor>();  
        if (customPreCommandInterceptors!=null) {  
            commandInterceptors // 自定义的前置拦截器  
                .addAll(customPreCommandInterceptors);  
        }  
        commandInterceptors // 默认的命令拦截器  
            .addAll(getDefaultCommandInterceptors());  
        if (customPostCommandInterceptors!=null) {  
            commandInterceptors // 自定义的后置拦截器  
                .addAll(customPostCommandInterceptors);  
        }  
        // 最后添加命令调用者CommandInvoker  
        commandInterceptors.add(new CommandInvoker());  
    }  
}
```

上面的代码用于设置所有的拦截器，这包括用户自定义的前置拦截器、默认拦截器、用户自定义的后置拦截器，以及命令调用拦截器，其中CommandInvoker是最终调用命令的处理类，在调用命令时会同时把CommandContext对象传递给具体的命令处理类。

所有的拦截器均实现接口CommandInterceptor，该接口的定义如下：

```
public interface CommandInterceptor {  
    <T> T execute(CommandConfig config, Command<T> command);  
    CommandInterceptor getNext();  
    void setNext(CommandInterceptor next);  
}
```

接口CommandInterceptor提供了next参数的getter和setter方法，next参数为下一个拦截器的对象引用，使用过JAVAE中Filter链的读者可能马上就明白了这其中的奥妙了，N个拦截器通过next属性关联就形成了一个无限的拦截器链。现在有一个问题要抛出来，哪个是第一个拦截器呢？

在上面的getDefaultCommandInterceptors()方法中首先添加的是日志拦截器，所以日志拦截器会作为第一个拦截器被执行，如果有事务拦截器，那么会在日志拦截器执行完成后被调用，最后才是命令上下文拦截器。在ProcessEngineConfigurationImpl类中设置断点后可以看到如图21-9所示的结果，很清晰地展示了拦截器的关联关系（先后顺序）。

图21-9中的“first”变量指向了日志拦截器LogInterceptor类，该拦截器会先输出“starting XxxCmd”，然后调用next变量指向拦截器的execute方法，依次循环直到所有的拦截器都执行完成。

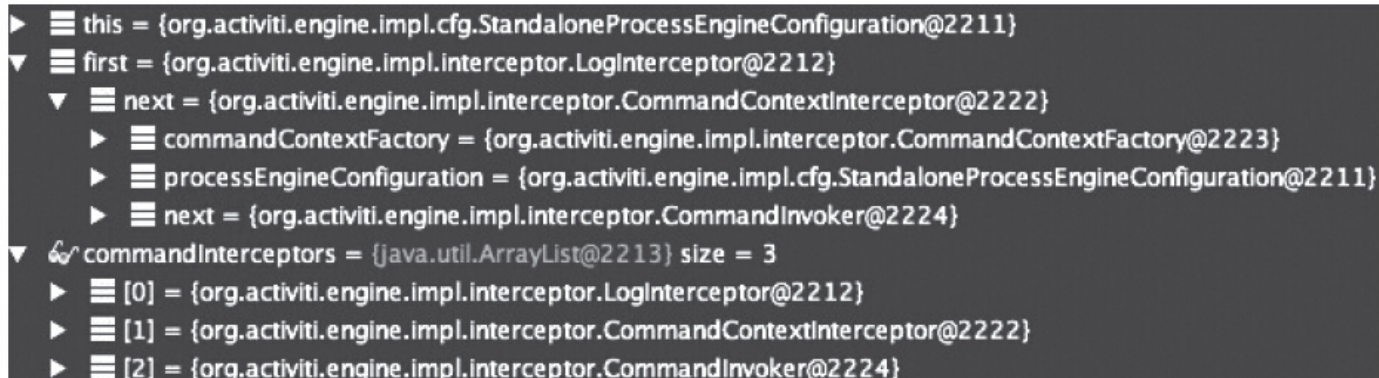


图21-9 在ProcessEngineConfigurationImpl类中断点查看拦截器顺序

了解了拦截器的串联关系，现在要思考另外一个问题：第一个拦截器如何被调用的呢？

在本节内容开头提到过的CommandExecutor类用来统一执行所有的命令，该类在初始化时把图21-9中的first变量作为构造参数传递，也就是由CommandExecutor类打开了命令执行的大门。下面的

代码取自源码中TaskServiceImpl类中的创建新任务方法，通过commandExecutor执行一个命令类NewTaskCmd。

```
public Task newTask(String taskId) {
    return commandExecutor.execute(new NewTaskCmd(taskId));
}
```

21.3.2 自定义命令

在了解了命令与拦截器的运行机制后我们可以自定义一个命令交给引擎执行，基于一整套完善的命令与拦截器运行机制可以很方便地实现自定义的命令，可以基于引擎原有功能进行扩展。

代码清单21-16列出了一个自定义命令，实现了更改任务名称的简单功能，然后在代码清单21-17中执行自定义命令。

代码清单21-16 自定义命令——更改任务名称

```
public class ModifyTaskNameCmd implements Command<Task> {
    protected String taskId;
    protected String taskName;
    public ModifyTaskNameCmd(String taskId, String taskName) {
        this.taskId = taskId;
        this.taskName = taskName;
    }
    @Override
    public Task execute(CommandContext commandContext) {
        // 获取任务实体管理器，根据任务ID查询任务实体对象
        TaskEntityManager taskEntityManager = commandContext.getTaskEntityManager();
        TaskEntity task = taskEntityManager.findTaskById(taskId);
        task.setName(taskName);
        task.update(); // 写回到数据库
        return task;
    }
}
```

代码清单21-17 执行自定义命令

```
@Test
public void testCustomCommand() throws Exception {
    initProcessEngine("chapter21/activiti.cfg.chapter21.command.xml");
    deploy("chapter21/leave.bpmn");
    ProcessInstance processInstance = startProcess();
    // 验证任务名称
    Task task = taskService.createTaskQuery()
        .processInstanceId(processInstance.getId()).singleResult();
    assertEquals("部门领导审批", task.getName());
    managementService.executeCommand( // 执行自定义的命令
        new ModifyTaskNameCmd(task.getId(), task.getName() + "-Modified"));
    task = taskService.createTaskQuery().singleResult();
    assertEquals("部门领导审批-Modified", task.getName());
}
```

在代码清单21-17的单元测试代码中首先启动了流程，接着验证待力任务的名称是否为“部门领导审批”，然后执行代码清单21-16的自定义命令，最后验证结果是否正确。

代码清单21-16是一个简单的命令实现，目的是让读者快速了解命令的定义及执行。而代码清单21-18则是一个实用的命令实现，可以不经过预设的输出流直接跳跃到其他节点，也就是常说的“任意节点跳转”功能。命令JumpActivityCmd使用了引擎内部未公开的API实现节点跳转动作，本节不做详细介绍，作为一般了解内容即可。

代码清单21-18 节点跳转命令JumpActivityCmd

```
public class JumpActivityCmd implements Command<ExecutionEntity> {
    private String activityId;
    private String processInstanceId;
    public JumpActivityCmd(String processInstanceId, String activityId) {
        this.activityId = activityId;
        this.processInstanceId = processInstanceId;
    }
    public ExecutionEntity execute(CommandContext commandContext) {
        // 查询活动的分支
        ExecutionEntity executionEntity = commandContext.getExecutionEntityManager()
            .findExecutionById(processInstanceId);
        executionEntity.destroyScope("jump"); // jump参数为销毁的原因
        // 从流程定义中查询目标Activity
        ProcessDefinitionImpl processDefinition = executionEntity.getProcessDefinition();
        ActivityImpl activity = processDefinition.findActivity(activityId);
        executionEntity.executeActivity(activity); // 节点跳转到目标活动
        return executionEntity;
    }
}
```

代码清单21-19的单元测试用来执行代码清单21-18的JumpActivityCmd命令，并且验证该命令是否发挥了应有的作用。

代码清单21-19 测试节点跳转命令JumpActivityCmd

```
@Test
public void testCustomCommandJump() throws Exception {
    initProcessEngine("chapter21/activiti.cfg.chapter21.command.xml");
    deploy("chapter21/leave.bpmn");
    ProcessInstance processInstance = startProcess();
    // 验证任务名称
    Task task = taskService.createTaskQuery().singleResult();
    assertEquals("部门领导审批", task.getName());
    managementService.executeCommand( // 直接跳转到销假节点
        new JumpActivityCmd(processInstance.getId(), "reportBack"));
    task = taskService.createTaskQuery().singleResult();
    assertEquals("销假", task.getName());
}
```

代码清单21-19单元测试执行成功，表示任务节点被跳转到了“销假”节点，中间跳过了“人事审批”，这个功能可以在特殊情况下使用。

21.3.3 命令拦截器

在21.3.1小节中详细介绍了命令与拦截器的执行机制，并且提到了自定义命令拦截器（分为前置和后置），本小节将介绍如何定义、配置自定义的命令拦截器。

我们分别定义两个类型的命令拦截器来演示前置与后置拦截器，代码清单21-20列出了前置拦截器类MyPreCommandInterceptor，代码清单21-21列出了后置拦截器类MyPostCommandInterceptor，在这两个拦截器中简单打印了被拦截的命令。

代码清单21-20 前置命令拦截器


```
public class MyPreCommandInterceptor extends AbstractCommandInterceptor {
    @Override
    public <T> T execute(CommandConfig config, Command<T> command) {
        System.out.println("前置命令拦截器->: " + command);
        return next.execute(config, command); // 继续执行下一个拦截器
    }
}
```

代码清单21-21 后置命令拦截器

```
public class MyPostCommandInterceptor extends AbstractCommandInterceptor {
    @Override
    public <T> T execute(CommandConfig config, Command<T> command) {
        System.out.println("后置命令拦截器->: " + command);
        return next.execute(config, command); // 继续执行下一个拦截器
    }
}
```

自定义拦截器需要在引擎配置对象中定义，下面的代码在引擎配置对象中分别为前置和后置命令拦截器集合注入了自定义命令拦截器实现类。

```
<bean id="processEngineConfiguration" class="org.activiti.enginehttp://www.hzcourse.com/resource/readBook?path=/openresources/teach_ebook/uncompressed/15030/OEBPS/Text/..XxxProc
    <property name="databaseSchemaUpdate" value="true"/>
    <!-- 前置命令拦截器集合属性 -->
    <property name="customPreCommandInterceptors">
        <list>
            <bean class="me.kafeitu.activiti.chapter21
                .command.MyPreCommandInterceptor"></bean>
        </list>
    </property>
    <!-- 后置命令拦截器集合属性 -->
    <property name="customPostCommandInterceptors">
        <list>
            <bean class="me.kafeitu.activiti.chapter21
                .command.MyPostCommandInterceptor"></bean>
        </list>
    </property>
</bean>
```

现在再执行代码清单21-19的单元测试，观察控制台（过滤字符“拦截器”）可以看到如下的输出：

```
前置命令拦截器->: org.activiti.engine.impl.SchemaOperationsProcessEngineBuild@904eabb
后置命令拦截器->: org.activiti.engine.impl.SchemaOperationsProcessEngineBuild@904eabb
前置命令拦截器->: org.activiti.engine.impl.cmd.DeployCmd@43277a30
前置命令拦截器->: org.activiti.engine.impl.cmd.DeployCmd@43277a30
前置命令拦截器->: org.activiti.engine.impl.cmd.GetNextIdBlockCmd@56ce3b62
前置命令拦截器->: org.activiti.engine.impl.cmd.GetNextIdBlockCmd@56ce3b62
前置命令拦截器->: org.activiti.engine.impl.ProcessDefinitionQueryImpl@2d97d09f
前置命令拦截器->: org.activiti.engine.impl.ProcessDefinitionQueryImpl@2d97d09f
前置命令拦截器->: org.activiti.engine.impl.cmd.SubmitStartFormCmd@2d6f4ce0
前置命令拦截器->: org.activiti.engine.impl.cmd.SubmitStartFormCmd@2d6f4ce0
前置命令拦截器->: org.activiti.engine.impl.TaskQueryImpl@758c3b7
前置命令拦截器->: org.activiti.engine.impl.TaskQueryImpl@758c3b7
前置命令拦截器->: me.kafeitu.activiti.chapter21.command.JumpActivityCmd@7aa36771
前置命令拦截器->: me.kafeitu.activiti.chapter21.command.JumpActivityCmd@7aa36771
前置命令拦截器->: org.activiti.engine.impl.TaskQueryImpl@1e605b1e
前置命令拦截器->: org.activiti.engine.impl.TaskQueryImpl@1e605b1e
```

21.4 流程虚拟机——PVM

Java语言是跨平台的，这得益于强大的JVM（Java Virtual Machine，即Java虚拟机），它是一个使用软件技术虚构出来的一台计算机，提供了一套计算有关规范，不同的操作系统按照规范有不同的实现方式，但是对外接受计算请求有统一的接口。

PVM（Process Virtual Machine，即流程虚拟机），是Activiti整个流程驱动（也叫流程推进）机制的核心规范。和JVM类似，PVM是规范而不是具体的实现。PVM最早在jBPM4时提出，也是jBPM4作者Tom Baeyens（当时就职于JBoss公司）推崇的规范。遗憾的是jBPM5放弃了PVM而选择使用Drools Flow开发新的流程引擎，可能这也是导致Tom Baeyens离开JBoss后基于PVM开发一套新的工作流引擎的原因之一。

21.4.1 简述PVM

在Activiti源码中针对PVM的规范定义了一些接口以及抽象类，Activiti实现BPMN 2.0规范中的元素全部基于PVM的这些抽象类，不同的元素虽然拥有相同的命令接口但是处理命令的方式不同，这样不同的元素实现由PVM串联成一个完整的流程。在Activiti中仅仅有PVM是不够的，不同的BPMN 2.0行为（Behavior）实现类还使用了Petri-Net（佩特里网）^[1]与UML Activity Graph两种数学与图形表示方式。

大家常用的Java虚拟机是由Oracle（原公司Sun被Oracle收购）公司开发的Oracle JVM，当然也有其他的JVM实现，例如OpenJDK提供的JVM。我们开发的一般应用可以在两种虚拟机上运行，这是因为两种虚拟机实现的规范大致相同。同理，PVM也可以做到通用，在Activiti中有一套基于PVM的实现，我们自己也可以基于PVM开发出自己的流程引擎，当然自己实现BPMN规范需要很大的工作量，这也是为什么Activiti到目前为止还没有完全覆盖BPMN 2.0规范所有元素的原因之一。

21.4.2 Hello PVM

虽然我们不太可能基于PVM开发出一套新的流程引擎，但是我们可以通过一些简单的代码来了解PVM是如何运行的，这样也可以更好地理解Activiti中流程驱动的深层次原理。在Activiti源码中有大量有关PVM的测试用例，这是我们学习和理解PVM的一个很好的途径，有兴趣的读者可以搜索源码中包含PVM的Java类。

很多读者对PVM还有些陌生，所以我们从最简单的示例开始讲解。图21-10展示了一个简单的流程图，包含开始、用户任务、结束三个流程活动。我们可以通过创建三种不同的活动对象分别处理各自的功能，最后调用PVM的API执行整个流程，具体的实现代码参见代码清单21-22。



图21-10 一个简单的流程图

代码清单21-22 Hello PVM

```

public class PvmTest {
    @Test
    public void helloPvm() {
        // 创建开始活动
        ActivityBehavior startBehavior = new ActivityBehavior() {           #1-S
            @Override
            public void execute(ActivityExecution execution) throws Exception {
                System.out.println("处理开始节点");                         #1-1
                List<PvmTransition> transitions = execution // 查询输出流
                    .getActivity().getOutgoingTransitions();
                execution.take(transitions.get(0)); // 转向输出流           #1-2
            }
        };
        // 创建用户任务活动
        ActivityBehavior userTaskBehavior = new ActivityBehavior() {         #2-S
            @Override
            public void execute(ActivityExecution execution) throws Exception {
                System.out.println("处理用户任务");
                List<PvmTransition> transitions = execution
                    .getActivity().getOutgoingTransitions();               #2-1
                execution.take(transitions.get(0));                         #2-E
            }
        };
        // 创建结束活动
        ActivityBehavior endBehavior = new ActivityBehavior() {             #3-S
            @Override
            public void execute(ActivityExecution execution) throws Exception {
                System.out.println("处理结束节点");
            }
        };
        // 创建流程定义构造器
        ProcessDefinitionBuilder builder = new ProcessDefinitionBuilder();   #4
        builder.createActivity("start") // 设置活动的名称                    #5-S
            .initial() // 标记为起始节点
            .behavior(startBehavior) // 活动处理器
            .transition("userTask") // 下一个执行的活动
            .endActivity(); // 结束当前活动                                  #5-E
        // 创建用户任务
        builder.createActivity("userTask").behavior(userTaskBehavior)        #6
            .transition("end").endActivity();
        // 创建结束节点
        builder.createActivity("end").behavior(endBehavior).endActivity();   #7
        // 构建PVM流程定义
        PvmProcessDefinition pvmProcessDefinition = builder.buildProcessDefinition();
        PvmProcessInstance processInstance = // 创建PVM流程实例              #8
            pvmProcessDefinition.createProcessInstance();                   #9-S
        processInstance.start(); // 启动流程                                #9-E
    }
}
  
```

代码清单21-22的#1、#2、#3处分别基于ActivityBehavior接口创建了三个匿名类，#1处的startBehavior用来处理图21-10中的“开始”活动，#2处的userTaskBehavior用来处理图21-10中的“用户任务”活动，#3处的endBehavior用来处理图21-10中的“结束”活动，其中#1-2、#2-1处从ActivityExecution对象中获取到当前活动的输出流（也可以称为转向或者连线），输出流可能会有多个，所以execution.getActivity()得到的是一个集合，这里为了简单起见只处理第一个输出流。当前活动的输出流是如何设置的呢？带着这个问题我们继续向下看代码。

#4处创建的ProcessDefinitionBuilder对象专门用来构建ProcessDefinition（流程定义），在执行完#5处的代码后就创建了一个名称为“start”的活动并声明这是起始节点，该活动的处理器设置为startBehavior对象，也就是说当流程运行到名称为start的节点时执行startBehavior类对象，执行完成后把输出流指向名称为“userTask”的活动。#6、#7处分别创建了用户任务和结束活动的属性，然后在#8处使用构造器构建一个PvmProcessDefiniton对象。这一过程与前面章节介绍的类似，使用流程设计器设计流程相当于代码清单21-22中#5、#6、#7处的流程活动关联配置，#8处构建流程定义对象相当于部署流程文件。

#9处则相当于调用了runtimeService.startProcessInstanceByXxx()方法启动一个流程返回一个流程实例。最后运行代码清单21-22中的单元测试，观察控制台的输出可以看到如下的输出：

```

处理开始节点
10:12:45,203 [main] DEBUG org.activiti.engine.impl.pvm.runtime.AtomicOperationTransitionNotifyListenerTake - ProcessInstance[1310188746] takes transition (start)--> (userTask)
10:12:45,204 [main] DEBUG org.activiti.engine.impl.pvm.runtime.AtomicOperation ActivityExecute - ProcessInstance[1310188746] executes Activity(userTask): me.kafeitu.activiti.chapter
处理用户任务
10:12:45,204 [main] DEBUG org.activiti.engine.impl.pvm.runtime.AtomicOperationTransitionNotifyListenerTake - ProcessInstance[1310188746] takes transition (userTask)-->(end)
10:12:45,205 [main] DEBUG org.activiti.engine.impl.pvm.runtime.AtomicOperation ActivityExecute - ProcessInstance[1310188746] executes Activity(end): me.kafeitu.activiti.chapter
处理结束节点
  
```

21.4.3 PVM进阶

执行代码清单21-22后启动了一个流程，接着控制台打印了三个活动的全部输出信息，按照我们在Activiti中设计的流程，应该是启动流程后停留在“用户任务”节点，等待用户触发后继续执行后续活动。

代码清单21-23定义了一个自动执行的活动类Automatic，代码清单21-24定义了一个等待类型的活动类WaitState。

代码清单21-23 自动执行活动实现类Automatic

```

public class Automatic implements ActivityBehavior {
    public void execute(ActivityExecution execution) throws Exception {
        List<PvmTransition> outgoingTransitions = execution
            .getActivity().getOutgoingTransitions();
        if (outgoingTransitions.isEmpty()) {
            execution.end(); // 如果没有后续的输出流，结束ActivityExecution
            System.out.println("流程已结束，结束节点: " + execution.getActivity().getId());
        } else {
            System.out.println("自动节点: " + execution.getActivity().getId());
            execution.take(outgoingTransitions.get(0));
        }
    }
}
  
```

代码清单22-23中处理了两种情况，如果当前活动还有后续输出流，则执行，否则结束ActivityExecution，根据活动是否并行、多实例判断是结束流程还是结束一个分支（Execution）。

代码清单21-24 支持等待功能的活动实现类WaitState

```
public class WaitState implements SignallableActivityBehavior {
    public void execute(ActivityExecution execution) throws Exception {
        System.out.println("节点" + execution.getActivity().getId() + "等待被触发");
    }
    public void signal(ActivityExecution execution, String signalName, Object signalData)
        throws Exception {
        PvmActivity activity = execution.getActivity();
        System.out.println("触发活动" + activity.getId());
        PvmTransition transition = activity.getOutgoingTransitions().get(0);
        execution.take(transition);
    }
}
```

代码清单22-24和代码清单22-23就有一些区别了，类WaitState实现的接口不是ActivityBehavior而是SignallableActivityBehavior，理解为可以接收信号的活动处理器。该接口继承自ActivityBehavior，定义了一个signal()方法，可以接受信号（signalName参数）和信号相关的数据（signalData参数）；从ActivityBehavior继承来的execute()方法用来处理具体的活动任务。

以图21-6的流程图为例，代码清单22-25列出了相关测试类，依赖了代码清单22-23和代码清单22-24的两个活动类。

代码清单21-25 测试等待发送信号的活动处理器

```
@Test
public void pvmSignalTask() {
    // 创建流程定义构造器
    ProcessDefinitionBuilder builder = new ProcessDefinitionBuilder();
    builder.createActivity("start") // 设置活动的名称
        .initial() // 标记为起始节点
        .behavior(new Automatic()) // 活动处理器
        .transition("userTask") // 下一个执行的活动
        .endActivity(); // 结束当前活动
    // 创建用户任务
    builder.createActivity("userTask").behavior(new WaitState())
        .transition("end").endActivity();
    // 创建结束节点
    builder.createActivity("end").behavior(new Automatic()).endActivity();
    // 构建PVM流程定义
    PvmProcessDefinition pvmProcessDefinition = builder.buildProcessDefinition();
    // 创建PVM流程实例
    PvmProcessInstance processInstance = pvmProcessDefinition.createProcessInstance();
    processInstance.start(); // 启动流程
    processInstance.signal("userTask", null); // 触发等待的活动（userTask）
}
```

在讲解代码清单22-25之前先查看一下运行结果，控制台的输出如下：

```
自动节点: start
节点userTask等待被触发
触发活动userTask
流程已结束，结束节点: end
```

在代码清单22-25中最关键的一行代码就是最后一行，调用了processInstance的signal()方法。读者可以把这一行代码注释掉再运行，结果如下所示，userTask一直等待被触发流程没有继续执行。

```
自动节点: start
节点userTask等待被触发
```

通过本小节的学习可以简单总结一下Activiti中BPMN 2.0规范的活动类是如何实现的：所有的自动节点（包括开启、结束、Service Task、输出流等）实现的ActivityBehavior接口被触发后直接把相应的任务处理完成，所有的等待节点（用户任务、中间捕获事件等）执行完ActivityBehavior接口的execute方法后并没有流转到下一节点，而是等待接收信号后才流转到下一节点，也就是调用SignallableActivityBehavior接口的signal方法。

[1] <http://zh.wikipedia.org/wiki/Petri网>

21.5 本章小结

前面所有章节都在讲解如何使用Activiti，而本章则深层次剖析Activiti是怎么运转的，这些内容能够让读者更加深入地了解Activiti的运行原理，我们把一个完整的引擎拆解开来摆在眼前，使大家一目了然。

本章从应用的角度开始，首先讲解了引擎如何解析BPMN文件，如何用代码的方式动态创建一个流程并启动，并且在引擎解析BPMN文件时通过注入的方式在拦截解析过程中加入自己的逻辑处理。全局事件是引擎留给我们的一个拦截运行过程的一个方便的入口，利用全局事件监听多种类型的事件，利用这些事件可以实现一些通用的功能或记录日志。

接着介绍了Activiti如何定义、使用命令与拦截器，详细介绍了引擎如何为命令类提供所需数据（例如数据源），以及如何添加自定义的拦截器到拦截器处理链中。

最后用读者熟知的JVM作为参考介绍了Activiti引擎的基础实现PVM（流程虚拟机），并通过示例了解其运作过程，以加深读者对流程驱动原理的理解。